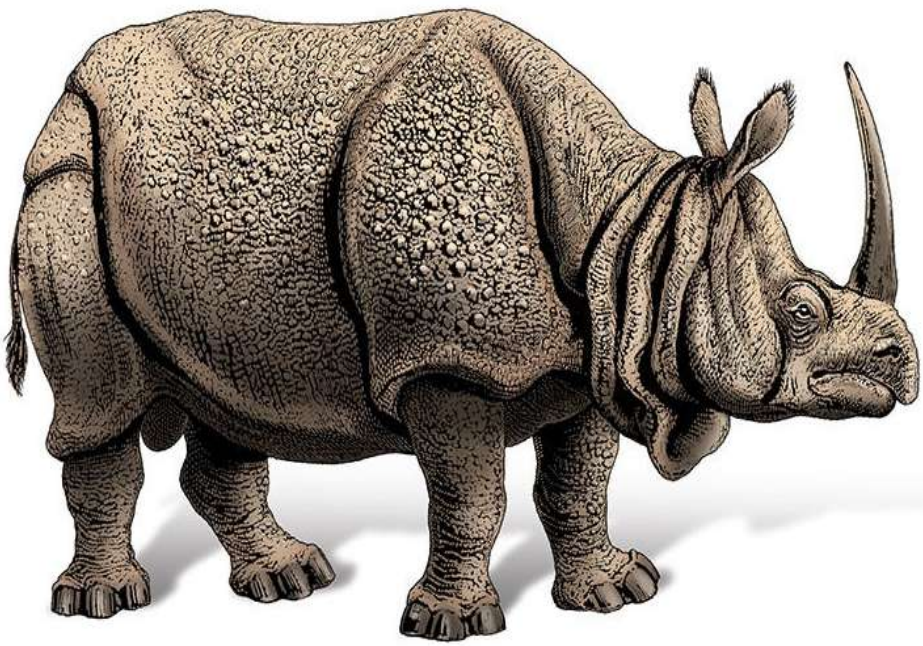
O'REILLY®

7-е издание

JavaScript Полное руководство

Справочник по самому популярному языку программирования



Дэвид Флэнаган

SEVENTH EDITION

JavaScript: The Definitive Guide

Master the World's Most-Used Programming Language

David Flanagan

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

СЕДЬМОЕ ИЗДАНИЕ

JavaScript Полное руководство

Справочник по самому популярному языку программирования

Дэвид Флэнаган

Диалектика

Москва · Санкт-Петербург
2021

ББК 32.973.26-018.2.75
Ф73
УДК 004.432

ООО "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info.dialektika@gmail.com, http://www.dialektika.com

Флэнаган, Дэвид.

Ф73 JavaScript. Полное руководство, 7-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2021. — 720 с. : ил. — Парал. тит. англ.

ISBN 978-5-907203-79-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Дэвид Флэнаган

JavaScript. Полное руководство
7-е издание

Подписано в печать 28.10.2020. Формат 70×100/16
Гарнитура Times
Усл. печ. л. 58,05. Уч.-изд. л. 45,7
Тираж 400 экз. Заказ № 7137.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-79-2 (рус.)

ISBN 978-1-491-95202-3 (англ.)

Оглавление

Содержание

Моим родителям, Донне и Мэтту,
с любовью и благодарностью.

Об авторе

Дэвид Флэнаган занимается программированием и пишет о JavaScript, начиная с 1995 года. Он живет с женой и детьми на Тихоокеанском Северо-Западе между городами Сиэтл и Ванкувер, Британская Колумбия. Дэвид получил диплом в области компьютерных наук и инженерии в Массачусетском технологическом институте и работает инженером-программистом в VMware.

Об иллюстрации на обложке

Животное, изображенное на обложке книги — это яванский носорог (лат. *Rhinoceros sondaicus*). Все пять видов носорогов характеризуются крупным размером, толстой бронеподобной кожей, трехпалыми лапами и одинарным или двойным носовым рогом. Яванский носорог имеет сходство с родственным ему индийским носорогом, и как у этого вида, самцы имеют одинарный рог. Однако яванские носороги меньше по размеру и обладают уникальной структурой кожи. Хотя в наши дни яванские носороги встречаются только в Индонезии, когда-то они обитали по всей Юго-Восточной Азии. Они живут в местах произрастания тропических лесов, где питаются обильной листвой и травой и прячутся от насекомых-вредителей, таких как кровососущие мухи, окуная свои морды в воду или грязь.

Яванский носорог имеет в среднем двухметровую высоту и может достигать трех и более метров в длину; взрослые особи весят до 1,4 тонны. Подобно индийским носорогам его серая кожа кажется разделенной на "пластины", некоторые из которых текстурированы. Естественная продолжительность жизни яванского носорога оценивается в 45–50 лет. Самки рожают каждые 3–5 лет после периода беременности, составляющего 16 месяцев. Детеныши при рождении весят около 45 килограмм и остаются под защитой своих матерей вплоть до двух лет.

Носороги в основном являются довольно многочисленным животным, которое способно адаптироваться к целому ряду сред обитания и во взрослом возрасте не имеет естественных хищников. Тем не менее, люди охотились на них почти до исчезновения. Бытовало верование в то, что рог носорога обладает магической и возбуждающей силой, из-за чего носороги стали главной целью браконьеров. Популяция яванского носорога наиболее нестабильна: по состоянию на 2020 год около 70 оставшихся особей этого вида живут под охраной в национальном парке Уджунг-Кулон на острове Ява в Индонезии. Похоже, что такая стратегия помогает обеспечить выживание этих носорогов на данный момент, т.к. в 1967 году их насчитывалось только 25.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой уничтожения; все они важны для нашего мира.

Предисловие

В настоящей книге рассматривается язык JavaScript и API-интерфейсы JavaScript, реализованные в веб-браузерах и Node. Я написал ее для читателей, которые имеют определенный опыт программирования и хотят изучить JavaScript, а также для программистов, которые уже используют JavaScript, но стремятся повысить уровень своих знаний и мастерства владения языком. Моя цель в этой книге — исчерпывающе и всесторонне документировать язык JavaScript, чтобы предоставить подробное введение в наиболее важные API-интерфейсы стороны клиента и сервера, доступные программам на JavaScript. В результате получилась толстая и подробная книга. Однако я надеюсь, что вы будете вознаграждены за ее тщательное изучение, а время, потраченное на чтение, будет скомпенсировано в форме более высокой продуктивности программирования.

Предыдущие издания книги содержали всесторонний справочный раздел. Я больше не считаю, что имеет смысл включать такой материал в печатный вариант, когда настолько легко и быстро найти актуальную справочную информацию в Интернете. Если вас интересует что-то связанное с JavaScript стороны клиента, то я рекомендую посетить веб-сайт MDN (`https://developer.mozilla.org`), а за сведениями об API-интерфейсах Node стороны сервера обращаться напрямую к источнику и просматривать справочную документацию Node.js (`https://nodejs.org/api`).

Соглашения, используемые в этой книге

В книге приняты следующие типографские соглашения.

Курсив

Применяется для акцентирования внимания и для новых терминов.

`Моноширинный`

Применяется для всего кода JavaScript, CSS-стилей и HTML-разметки, а также в целом для всего, что вы будете набирать буквально при программировании.

Здесь приводится общее замечание.

Здесь приводится предупреждение или предостережение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и т.п.) для этой книги доступны для загрузки по ссылке `https://github.com/davidflanagan/jstdg7` или на веб-странице русского издания книги по адресу: `http://www.williamspublishing.com/Books/978-5-907203-79-2.html`.

Данная книга призвана помочь вам делать свою работу. В общем случае вы можете применять приведенный здесь код примеров в своих программах и документации. Вы не обязаны спрашивать у нас разрешения, если только не воспроизводите значительную часть кода. Например, для написания программы, в которой встречаются многие фрагменты кода из этой книги, разрешение не требуется. Тем не менее, для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение обязательно. Ответ на вопрос путем ссылки на эту книгу и цитирования кода из примера разрешения не требует. Но для внедрения существенного объема кода примеров, предлагаемых в этой книге, в документацию по вашему продукту разрешение обязательно.

Мы высоко ценим указание авторства, хотя и не требуем этого. Установление авторства обычно включает название книги, фамилии и имена авторов, издательство и номер ISBN. Например: "*JavaScript: TheDefinitive Guide, Seventh Edition*, by David Flanagan (O'Reilly). Copyright 2020 David Flanagan, 978-1-491-95202-3".

Если вам кажется, что способ использования вами примеров кода выходит за законные рамки или упомянутые выше разрешения, тогда свяжитесь с нами по следующему адресу электронной почты: `permissions@oreilly.com`.

Благодарности

Появлению настоящей книги содействовали многие люди. Я хочу поблагодарить моего редактора, Анжелу Руфино, за то, что помогала мне придерживаться графика, и за ее терпеливое отношение к пропущенным мною срокам. Спасибо также моим техническим рецензентам: Брайану Слеттену, Элизабет Робсон, Итану Флэнагану, Максимилиану Фиртману, Саре Вакс и Шалку Нитлингу. Их комментарии и предложения сделали эту книгу лучше.

Производственная команда в O'Reilly отлично выполнила свою обычную работу: Кристен Браун руководила производственным процессом, Дебора Бейкер

была редактором, Ребекка Демарест вычерчивала рисунки, а Джуди Мак-Федрис создавала предметный указатель.

В число редакторов, рецензентов и участников предыдущих изданий книги входили: Эндрю Шульман, Анджело Сиригос, Аристотель Пагальцис, Брендан Эйх, Кристиан Хилманн, Дэн Шафер, Дэйв Митчелл, Деб Кэмерон, Дуглас Крокфорд, доктор Танкред Хиршманн, Дилан Шиманн, Фрэнк Уиллисон, Джефф Стернс, Герман Вентер, Джей Ходжес, Джефф Йейтс, Джозеф Кессельман, Кен Купер, Ларри Салливан, Линн Роллинз, Нил Беркман, Майк Лукидес, Ник Томпсон, Норрис Бойд, Пола Фергюсон, Питер-Поль Кох, Филипп Ле Хегарэ, Рафаэле Чекко, Ричард Якер, Сандерс Кляйнфельд, Скотт Фурман, Скотт Айзекс, Шон Катценбергер, Терри Аллен, Тодд Дитчендорф, Видур Аппарао, Вольдемар Хорват и Захари Кессин.

Написание седьмого издания держало меня вдали от семьи в течение многих поздних вечеров. Моя любовь к ним и благодарность за то, что они терпели мое отсутствие.

Дэвид Флэнаган, март 2020 года

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`
WWW: `http://www.dialektika.com`

ГЛАВА 1

Введение в JavaScript

JavaScript является языком программирования веб-сети. Подавляющее большинство веб-сайтов использует JavaScript, а все современные веб-браузеры — в настольных компьютерах, планшетах и смартфонах — включают интерпретаторы JavaScript, делая JavaScript самым распространенным языком программирования в истории. За последнее десятилетие исполняющая среда Node.js открыла возможность программирования на JavaScript за пределами веб-браузеров, и феноменальный успех Node означает, что JavaScript теперь также представляет собой наиболее часто применяемый язык программирования в сообществе разработчиков программного обеспечения (ПО). Независимо от того, начинаете вы с нуля или уже профессионально использовали JavaScript, книга позволит вам мастерски овладеть языком.

Если вы знакомы с другими языками программирования, то это поможет вам понять, что JavaScript — высокоуровневый, динамический, интерпретируемый язык программирования, который хорошо подходит для объектно-ориентированного и функционального стилей программирования. Переменные JavaScript являются нетипизированными. Синтаксис в определенной степени основан на Java, но в остальном языки не связаны. JavaScript получает свои первоклассные функции от языка функционального программирования Scheme, а основанное на прототипах наследование — от малоизвестного языка объектно-ориентированного программирования Self. Но для чтения этой книги и изучения JavaScript вам не нужно знать упомянутые языки или быть знакомым с их элементами.

Название "JavaScript" довольно обманчиво. За исключением кажущегося синтаксического сходства JavaScript совершенно отличается от языка программирования Java. К тому же JavaScript давно избавился от своих корней языка написания сценариев, чтобы стать надежным и эффективным языком, подходящим для серьезной разработки ПО и проектов с гигантскими базами исходного кода.

JavaScript: названия, версии и режимы

Язык JavaScript был создан в компании Netscape Communications на заре истории веб-сети. Формально "JavaScript" — это торговая марка, зарегистрированная компанией Sun Microsystems (теперь Oracle), которая применяется для описания реализации языка от Netscape (в настоящее время Mozilla). Компания Netscape представила язык для стандартизации Европейской Ассоциации производителей компьютеров (European Computer Manufacturer's Association — ECMA) и по причине сложностей с торговой маркой стандартизированной версии языка было навязано неуклюжее название "ECMAScript". На практике все называют язык просто JavaScript. Для ссылки на стандарт языка и на его версии в книге используется название "ECMAScript" и аббревиатура "ES".

На протяжении большей части 2010-х годов все веб-браузеры поддерживали версию 5 стандарта ECMAScript. В этой книге ES5 рассматривается как отправная точка совместимости и предшествующие ей версии языка больше не обсуждаются. Стандарт ES6 был выпущен в 2015 году и обзавелся важными новыми средствами, включая синтаксис классов и модулей, которые превратили JavaScript из языка написания сценариев в серьезный универсальный язык, подходящий для крупномасштабной разработки ПО. Начиная с ES6, спецификация ECMAScript перешла на ежегодный ритм выпусков и теперь версии языка — ES2016, ES2017, ES2018, ES2019 и ES2020 — идентифицируются по году выхода.

По мере развития JavaScript проектировщики языка пытались устранить дефекты в ранних версиях (до ES5). Ради поддержки обратной совместимости невозможно удалить устаревшие функции, какими бы дефектными они ни были. Но в ES5 и последующих версиях программы могут выбирать *строгий (strict) режим* JavaScript, в котором был исправлен ряд ранних языковых ошибок. Механизм включения представляет собой директиву `use strict`, описанную в подразделе 5.6.3, где также обобщаются отличия между устаревшим JavaScript и строгим JavaScript. В ES6 и последующих версиях применение новых функциональных средств часто неявно вызывает строгий режим. Например, если вы используете ключевое слово `class` из ES6 или создаете модуль ES6, тогда весь код внутри класса или модуля автоматически становится строгим, и в таких контекстах старые, дефектные функции не будут доступными. В книге раскрываются устаревшие функциональные средства JavaScript, но важно помнить о том, что в строгом режиме они не доступны.

Чтобы быть полезным, каждый язык обязан иметь платформу, или стандартную библиотеку, для выполнения таких действий, как базовый ввод и вывод. В основном языке JavaScript определен минимальный API-интерфейс для чисел, текста, массивов, множеств, отображений и т.д., но какая-либо функциональность ввода или вывода отсутствует. За ввод и вывод (а также за более сложные средства наподобие работы с сетью, хранилищем и графикой) несет ответственность "среда размещения", внутрь которой встраивается JavaScript.

Первоначальной средой размещения для JavaScript был веб-браузер, и он по-прежнему является самой распространенной исполняющей средой для кода JavaScript. Среда веб-браузера позволяет коду JavaScript получать ввод от мыши и клавиатуры пользователя и отправлять HTTP-запросы. Вдобавок она разрешает коду JavaScript отображать вывод пользователю с помощью HTML и CSS.

Начиная с 2010 года, для кода JavaScript стала доступной еще одна среда размещения. Вместо того, чтобы ограничивать JavaScript работой с API-интерфейсами, предоставляемыми веб-браузером, Node дает JavaScript доступ к целой операционной системе, позволяя программам JavaScript читать и записывать в файлы, посылать и получать данные по сети, а также отправлять и обслуживать HTTP-запросы. Node — популярный вариант для реализации веб-серверов и удобный инструмент для написания простых служебных сценариев как альтернативы сценариям командной оболочки.

Большая часть этой книги сосредоточена на самом языке JavaScript. В главе 11 описана стандартная библиотека JavaScript, в главе 15 представлена среда размещения веб-браузера, а в главе 16 обсуждается среда размещения Node.

В книге сначала раскрываются низкоуровневые основы, на которых затем строятся более развитые и высокоуровневые абстракции. Главы лучше читать более или менее в порядке их следования в книге. Но изучение нового языка программирования никогда не выглядело как линейный процесс, да и описание языка тоже нелинейно: каждое языковое средство связано с другими средствами, потому книга полна перекрестных ссылок — иногда назад, а временами вперед — на связанные материалы. В этой вводной главе предлагается краткий обзор языка, представляющий ключевые средства, который облегчит понимание их углубленного рассмотрения в последующих главах. Если вы уже являетесь практикующим программистом на JavaScript, тогда вполне можете пропустить текущую главу. (Хотя до того, как двигаться дальше, вы можете насладиться чтением примера 1.1 в конце главы.)

1.1. Исследование JavaScript

При изучении нового языка программирования важно опробовать примеры в книге, затем модифицировать их и снова испытать, чтобы проверить знание языка. Для этого вам необходим интерпретатор JavaScript.

Простейший способ испытания нескольких строк кода JavaScript предусматривает открытие инструментов разработчика вашего веб-браузера (нажав <F12>, <Ctrl+Shift+I> или <Command+Option+I>) и выбор вкладки Консоль. Далее вы можете набирать код в приглашении на ввод и по мере набора видеть результаты. Инструменты разработчика браузера часто появляются в виде панелей в нижней или правой части окна браузера, но обычно вы можете открепить их, сделав отдельными окнами (рис. 1.1), что часто действительно удобно.

Еще один способ опробовать код JavaScript заключается в загрузке и установке Node из `https://nodejs.org/`. После установки Node в системе вы можете просто открыть окно терминала и ввести `node`, чтобы начать интерактивный сеанс JavaScript вроде показанного ниже:

```
$ node
Welcome to Node.js v12.13.0.
Type ".help" for more information.
```
Добро пожаловать в Node.js v12.13.0.
Введите .help для получения дополнительной информации.
```
> .help
.break  Sometimes you get stuck, this gets you out
.clear  Alias for .break
.editor Enter editor mode
.exit   Exit the repl
.help   Print this help message
.load   Load JS from a file into the REPL session
.save   Save all evaluated commands in this REPL session to a file
```
.break Когда вы застряли, это поможет выбраться
.clear Псевдоним для .break
.editor Вход в режим редактора
.exit   Выход из сеанса REPL
.help   Вывод этого справочного сообщения
.load   Загрузка кода JS из файла в сеанс REPL
.save   Сохранение всех команд, выполненных в этом сеансе REPL, в файл
```
Press ^C to abort current expression, ^D to exit the repl
```
Нажмите ^C для прекращения текущего выражения, ^D для выхода из сеанса REPL
```
> let x = 2, y = 3;
undefined
```
не определено
```
> x + y
5
> (x === 2) && (y === 3)
true
> (x > 3) || (y < 3)
false
```

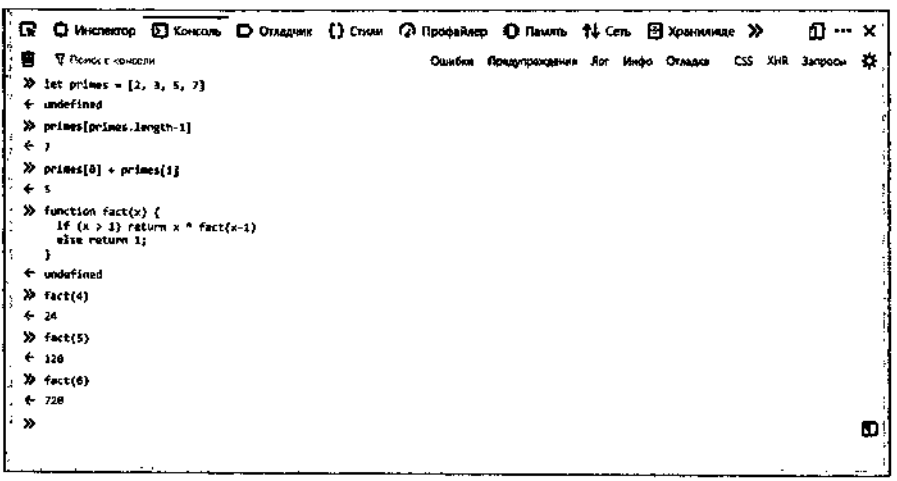


***Рис. 1.1.** Консоль JavaScript в инструментах разработчика Firefox*

1.2. Программа "Hello World"

Когда вы будете готовы приступить к экспериментированию с более длинными порциями кода, такие построчные интерактивные среды могут оказаться неподходящими, и вы вполне вероятно предпочтете набирать свой код в текстовом редакторе. Оттуда вы можете копировать и вставлять код в консоль JavaScript или в сеанс Node. Либо вы можете сохранить свой код в файл (с традиционным для кода JavaScript файловым расширением `.js`) и затем запустить такой файл кода JavaScript с помощью Node:

```
$ node snippet.js
```

Если вы применяете среду Node в подобной неинтерактивной манере, тогда она не будет автоматически выводить значение всего запущенного вами кода, так что вам придется делать это самостоятельно. Для отображения текста и других значений JavaScript в окне терминала или в консоли инструментов разработчика браузера вы можете использовать функцию `console.log()`. Таким образом, например, если вы создадите файл `hello.js`, содержащий следующую строку кода:

```
console.log("Hello World!");
```

и выполните его посредством `node hello.js`, то увидите выведенное сообщение `Hello World!`.

Если вы хотите, чтобы то же самое сообщение выводилось в консоли JavaScript веб-браузера, тогда создайте новый файл по имени `hello.html` и поместите в него такой текст:

```
<script src="hello.js"></script>
```

Затем загрузите `hello.html` в своем веб-браузере с применением URL вида `file://` наподобие:

```
file:///Users/username/javascript/hello.html
```

Открыв окно инструментов разработчика, вы увидите на консоли сообщение `Hello World!`.

1.3. Тур по JavaScript

В этом разделе представлено краткое введение в язык JavaScript через примеры кода. После такой вводной главы мы перейдем к самому низкому уровню JavaScript: в главе 2 будут объясняться такие элементы JavaScript, как комментарии, точки с запятой и набор символов Unicode, а в главе 3 мы рассмотрим более интересные темы — переменные JavaScript и значения, которые можно им присваивать.

Вот пример кода, иллюстрирующий основные моменты из указанных двух глав:

```
// Все, что следует за двумя символами косой черты, является комментарием.
// Читайте комментарии внимательно: в них объясняется код JavaScript.
```

```
// Переменная представляет собой символическое имя значения.
// Переменные объявляются с помощью ключевого слова let:
let x;                  // Объявление переменной по имени x.

// Присваивать значения переменным можно посредством знака =.
x = 0;                  // Теперь переменная x имеет значение 0.
x                       // => 0: переменная оценивается в свое значение.

// JavaScript поддерживает несколько типов значений.
x = 1;                  // Числа.
x = 0.01;               // Числа могут быть целыми или вещественными.
x = "hello world";      // Строки текста в кавычках.
x = 'JavaScript';       // Одинарные кавычки тоже устанавливают
                        // границы строк.
x = true;               // Булевское значение.
x = false;              // Другое булевское значение.
x = null;               // null - это специальное значение, которое
                        // означает отсутствие значения.
x = undefined;          // undefined - еще одно специальное значение,
                        // подобное null.
```

Программы JavaScript могут манипулировать еще двумя очень важными *типами* — объектами и массивами. Они обсуждаются в главах 6 и 7, но настолько важны, что вы будете многократно встречать их, прежде чем доберетесь до этих глав:

```
// Самым важным типом данных JavaScript является объект.
// Объект - это коллекция пар имя/значение
// или отображений строки на значение.
let book = {            // Объекты заключаются в фигурные скобки.
   topic: "JavaScript", // Свойство topic имеет значение JavaScript.
   edition: 7           // Свойство edition имеет значение 7.
};                      // Фигурная скобка помечает конец объекта.

// Доступ к свойствам объекта осуществляется с помощью . или []:
book.topic                      // => "JavaScript"
book["edition"]                 // => 7: еще один способ доступа
                                //       к значениям свойств.
book.author = "Flanagan";       // Новые свойства создаются
                                // посредством присваивания.
book.contents = {};     // {} - пустой объект, не имеющий свойств.

// Условный доступ к свойствам с помощью ?. (ES2020):
book.contents?.ch01?.sect1      // => не определено: book.contents
                                // не имеет свойства ch01.

// JavaScript также поддерживает массивы (списки с числовой
индексацией) значений:
let primes = [2, 3, 5, 7];      // Массив из 4 значений,
                                // ограниченный посредством [ и ].
primes[0]               // => 2: первый элемент (с индексом 0) массива.
primes.length           // => 4: количество элементов в массиве.
primes[primes.length-1]         // => 7: последний элемент массива.
primes[4] = 9;                  // С помощью присваивания можно
                                // добавлять новые элементы.
```

```
primes[4] = 11;                 // С помощью присваивания можно
                                // изменять существующие элементы.
let empty = [];                 // [] - пустой массив, не имеющий элементов.
empty.length                    // => 0

// Массивы и объекты могут содержать другие массивы и объекты:
let points = [                  // Массив с 2 элементами.
   {x: 0, y: 0},                // Каждый элемент является объектом.
   {x: 1, y: 1}
];
let data = {                    // Объект с двумя свойствами.
   trial1: [[1,2], [3,4]],      // Значением каждого свойства является массив.
   trial2: [[2,3], [4,5]]       // Элементы массива представляют собой массивы.
};
```

Синтаксис комментариев в примерах кода

Возможно, в предыдущем коде вы заметили, что некоторые комментарии начинаются с символов =>. Они показывают значение, выдаваемое кодом, и являются моей попыткой эмуляции в печатном издании интерактивной среды JavaScript вроде консоли веб-браузера.

Такие комментарии `//  =>` также служат *утверждениями*, и я написал инструмент, который тестирует код и проверяет, выдает ли он значения, указанные в комментариях. Я надеюсь, что это поможет сократить количество ошибок в книге.

Существуют два связанных стиля комментариев/утверждений. Если вы видите комментарий в форме `//  a  ==  42`, то он означает, что после выполнения кода, предшествующего комментарию, переменная a будет иметь значение 42. Если вы видите комментарий в форме `//  !`, то он означает, что код в строке перед комментарием генерирует исключение (после восклицательного знака в комментарии обычно объясняется вид генерируемого исключения).

Вы будете встречать описанные комментарии повсюду в книге.

Проиллюстрированный здесь синтаксис для перечисления элементов массива внутри квадратных скобок или отображения имен свойств объекта на значения свойств внутри фигурных скобок известен как *выражения инициализации* и будет рассматриваться в главе 4. *Выражение* представляет собой фразу JavaScript, которая может *вычисляться* для выдачи значения. Например, использование . и [] для ссылки на значение свойства объекта или элемента массива является выражением.

Один из наиболее распространенных способов формирования выражений в JavaScript предусматривает применение *операций*:

```
// Операции осуществляют действие над значениями (операндами),
// чтобы произвести новое значение.
// Арифметические операции относятся к наиболее простым:
3 + 2                   // => 5: сложение
3 - 2                   // => 1: вычитание
```

```
3 * 2                   // => 6: умножение
3 / 2                   // => 1.5: деление
points[1].x - points[0].x  // => 1: более сложные операнды тоже работают
"3" + "2"               // => "32": + выполняет сложение чисел
                        //          и конкатенацию строк

// В JavaScript определен ряд сокращенных арифметических операций:
let count = 0;          // Определение переменной
count++;                // Инкрементирование переменной
count--;                // Декрементирование переменной
count += 2;             // Прибавление 2: то же, что и count = count + 2;
count *= 3;             // Умножение на 3: то же, что и count = count * 3;
count                   // => 6: имена переменных - тоже выражения.

// Операции сравнения проверяют, равны или не равны два значения,
// больше или меньше одно из них и т.д. Результатом будет true или false:
let x = 2, y = 3;       // Знаки = здесь являются присваиванием,
                        // не проверками на равенство
x === y                 // => false: равенство
x !== y                 // => true: неравенство
x < y                   // => true: меньше
x <= y                  // => true: меньше или равно
x > y                   // => false: больше
x >= y                  // => false: больше или равно
"two" === "three"       // => false: две строки отличаются друг от друга
"two" > "three"         // => true: "tw" в алфавитном порядке больше, чем "th"
false === (x > y)       // => true: false равно false

// Логические операции объединяют или инвертируют булевские значения:
(x === 2) && (y === 3)  // => true: оба сравнения дают true.
                        //          && обозначает И
(x > 3) || (y < 3)      // => false: ни одно из сравнений не дает true.
                        //           || обозначает ИЛИ
!(x === y)              // => true: ! инвертирует булевское значение
```

Если выражения JavaScript подобны фразам, то *операторы* JavaScript похожи на полные предложения. Операторы будут обсуждаться в главе 5. Грубо говоря, выражение представляет собой что-то, что вычисляет значение, но ничего не *делает*: оно никак не изменяет состояние программы. С другой стороны, операторы не имеют значения, но изменяют состояние программы. Выше вы видели объявления переменных и операторы присваивания. Другой обширной категорией операторов являются *управляющие структуры*, такие как условные операторы и циклы. После рассмотрения функций будут приведены примеры.

Функция — это именованный и параметризованный блок кода JavaScript, который вы определяете один раз и затем можете вызывать снова и снова. Функции формально не раскрываются вплоть до главы 8, но подобно объектам и массивам вы будете их многократно видеть в главах, предшествующих главе 8.

Вот несколько простых примеров:

```
// Функции - это параметризованные блоки кода JavaScript,
// которые мы можем вызывать.
```

```
function plus1(x) {      // Определить функцию по имени plus1
                         // с параметром x.
   return x + 1;         // Возвратить значение на единицу
                         // больше переданного значения.
}                        // Функции заключаются в фигурные скобки.
plus1(y)                 // => 4: y равно 3, так что этот вызов
                         //       возвращает 3+1.
let square = function(x) {  // Функции являются значениями и могут
                            // присваиваться переменным.
   return x * x;         // Вычислить значение функции.
};                       // Точка с запятой помечает конец присваивания.
square(plus1(y))         // => 16: вызвать две функции в одном выражении.
```

В ES6 и последующих версиях имеется сокращенный синтаксис для определения функций. В этом кратком синтаксисе для отделения списка аргументов от тела функции используется =>, а потому функции, определенные подобным образом, известны как *стрелочные функции* (arrow function). Стрелочные функции чаще всего применяются, когда нужно передать неименованную функцию в качестве аргумента другой функции. Давайте перепишем предыдущий код с целью использования стрелочных функций:

```
const plus1 = x => x + 1;    // Вход x отображается на выход x + 1.
const square = x => x * x;   // Вход x отображается на выход x * x.
plus1(y)                     // => 4: вызов функции остается прежним.
square(plus1(y))             // => 16
```

Когда мы применяем функции с объектами, то получаем *методы*:

```
// Когда функции присваиваются свойствам объекта,
// мы называем их "методами".
// Методы имеют все объекты JavaScript (включая массивы):
let a = [];                  // Создать пустой массив.
a.push(1,2,3);               // Метод push() добавляет элементы в массив.
a.reverse();                 // Еще один метод: изменение порядка следования
                             // элементов на противоположный.

// Мы также можем определять наш собственный метод.
// Ключевое слово this ссылается на объект, в котором определяется
// метод: в данном случае на определенный ранее массив points.
points.dist = function() {   // Определить метод для расчета
                             // расстояния между точками.
   let p1 = this[0];         // Первый элемент массива, на котором вызван метод.
   let p2 = this[1];         // Второй элемент объекта this.
   let a = p2.x-p1.x;        // Разность координат x.
   let b = p2.y-p1.y;        // Разность координат y.
   return Math.sqrt(a*a + b*b);  // Теорема Пифагора.
                             // Math.sqrt() вычисляет квадратный корень.
};
points.dist() // => Math.sqrt(2): расстояние между нашими двумя точками.
```

А теперь, как и было обещано, рассмотрим несколько функций, в телах которых демонстрируются распространенные операторы JavaScript, относящиеся к управляющим структурам:

```
// JavaScript включает условные операторы и циклы, использующие
// синтаксис C, C++, Java и других языков.
function abs(x) {          // Функция для вычисления абсолютной величины.
   if (x >= 0) {           // Оператор if...
      return x;            // выполняет этот код, если результат
                           // сравнения истинный.
   }                       // Конец конструкции if.
   else {        // Необязательная конструкция else выполняет свой код,
      return -x;           // если результат сравнения ложный.
   }                       // Когда имеется один оператор на конструкцию,
                           // фигурные скобки необязательны.
}                          // Обратите внимание на операторы return,
                           // вложенные внутрь if/else.
abs(-10) === abs(10)       // => true

function sum(array) {      // Функция для расчета суммы элементов массива.
   let sum = 0;            // Начать с исходной суммы 0.
   for(let x of array){    // Пройти в цикле по массиву, присваивая
                           // каждый элемент переменной x.
      sum += x;            // Добавить значение элемента к сумме.
   }                       // Конец цикла.
   return sum;             // Возвратить сумму.
}
sum(primes)                // => 28: сумма первых 5 простых чисел 2+3+5+7+11.

function factorial(n) {// Функция для расчета факториалов.
   let product = 1;        // Начать с произведения 1.
   while(n > 1) {          // Повторять операторы в {} до тех пор,
                           // пока выражение в () истинно.
      product *= n;        // Сокращение для product = product * n;
      n--;                 // Сокращение для n = n - 1.
   }                       // Конец цикла.
   return product;         // Возвратить произведение.
}
factorial(4)               // => 24: 1*4*3*2

function factorial2(n) {// Еще одна версия с применением другого цикла.
   let i, product = 1;  // Начать с 1.
   for(i=2; i <= n; i++)// Автоматически инкрементировать i с 2 до n.
      product *= i;        // Каждый раз делать это. В однострочных
                           // циклах скобки {} не нужны.
   return product;         // Возвратить факториал.
}
factorial2(5)              // => 120: 1*2*3*4*5
```

Язык JavaScript поддерживает стиль объектно-ориентированного программирования, но он значительно отличается от стилей в "классических" языках объектно-ориентированного программирования. В главе 9 детально обсуждается объектно-ориентированное программирование в JavaScript и приводятся многочисленные примеры. Ниже показан очень простой пример, который демонстрирует, как определять класс JavaScript для представления двумерных геометрических точек. Объекты, являющиеся экземплярами этого класса, имеют

единственный метод по имени distance(), который рассчитывает расстояние от начала координат до точки:

```
class Point {                  // По соглашению имена классов начинаются
                               // с заглавной буквы.
    constructor(x, y) {        // Функция конструктора для инициализации
                               // новых экземпляров.
        this.x = x;            // Ключевое слово this - это новый
                               // инициализируемый объект.
        this.y = y;            // Сохранить аргументы функции как
                               // свойства объекта.
    }                          // Оператор return в функции конструктора
                               // не нужен.
    distance() {               // Метод для расчета расстояния от начала
                               // координат до точки.
        return Math.sqrt(      // Возвратить квадратный корень xI + yI.
            this.x * this.x + // this ссылается на объект Point,
            this.y * this.y    // на котором вызван метод distance.
        );
    }
}
// Использовать функцию конструктора Point() вместе с new
// для создания объектов Point:
let p = new Point(1, 1);      // Геометрическая точка (1,1).

// Теперь использовать метод объекта Point по имени p:
p.distance()                   // => Math.SQRT2
```

На этом вводный тур по фундаментальному синтаксису и возможностям JavaScript закончен, но далее в книге предлагаются самодостаточные главы, в которых раскрываются дополнительные средства языка.

Глава 10, "Модули"

Показывается, каким образом код JavaScript в одном файле или сценарии может пользоваться функциями и классами JavaScript, определенными в других файлах или сценариях.

Глава 11, "Стандартная библиотека JavaScript"

Рассматриваются встроенные функции и классы, которые доступны всем программам JavaScript. Сюда входят такие важные структуры данных, как отображения и множества, класс регулярных выражений для сопоставления с текстовыми шаблонами, функции для сериализации структур данных JavaScript и многое другое.

Глава 12, "Итераторы и генераторы"

Объясняется, каким образом работает цикл for/of и как можно делать собственные классы итерируемыми с помощью for/of. Также раскрываются генераторные функции и оператор yield.

Глава 13, "Асинхронный JavaScript"

Предлагается углубленное исследование асинхронного программирования на JavaScript с объяснением обратных вызовов и событий, API-интерфейсы на основе объектов `Promise` (обещаний или "промисов") и ключевых слов `async` и `await`. Хотя основной язык JavaScript не является асинхронным, в веб-браузерах и Node асинхронные API-интерфейсы применяются по умолчанию, поэтому в главе обсуждаются методики работы с ними.

Глава 14, "Метапрограммирование"

Представляется несколько развитых функциональных средств JavaScript, которые могут заинтересовать программистов, пишущих библиотеки кода для использования другими программистами на JavaScript.

Глава 15, "JavaScript в веб-браузерах"

Рассматривается среда размещения веб-браузера, объясняются особенности выполнения кода JavaScript веб-браузерами и раскрываются наиболее важные из многочисленных API-интерфейсов, определенных веб-браузерами. Это безоговорочно самая длинная глава в книге.

Глава 16, "JavaScript на стороне сервера с использованием Node"

Рассматривается среда размещения Node с исследованием фундаментальной программной модели, а также структур данных и API-интерфейсов, которые важнее всего понимать.

Глава 17, "Инструменты и расширения JavaScript"

Обсуждаются инструменты и языковые расширения, о которых полезно знать, поскольку они широко применяются и могут сделать вас более продуктивным программистом.

1.4. Пример: гистограмма частоты использования символов

Текущая глава завершается короткой, но нетривиальной программой JavaScript. Пример 1.1 представляет собой программу Node, которая читает текст из стандартного ввода, рассчитывает данные для гистограммы частоты применения символов во введенном тексте и выводит итоговую гистограмму. Вы могли бы запустить программу такого рода для анализа частоты использования символов в собственном исходном коде:

```
$ node charfreq.js < charfreq.js
T: ########### 11.22%
E: ########## 10.15%
R: ####### 6.68%
S: ###### 6.44%
A: ###### 6.16%
N: ###### 5.81%
```

```
O: ##### 5.45%
I: ##### 4.54%
H: #### 4.07%
C: ### 3.36%
L: ### 3.20%
U: ### 3.08%
/: ### 2.88%
```

В примере задействовано несколько развитых средств JavaScript и он предназначен для демонстрации того, как может выглядеть реальная программа JavaScript. Пока что вы не должны надеяться, что весь код окажется понятным, но будьте уверены — все будет объяснено в последующих главах.

Пример 1.1. Расчет данных для гистограмм частоты применения символов с помощью JavaScript

```
/**
 * Эта программа Node читает текст из стандартного ввода, рассчитывает
 * частоту встречи каждой буквы в данном тексте и отображает гистограмму
 * для наиболее часто используемых символов.
 * Для выполнения требует версии Node 12 или новее.
 *
 * В среде Unix программу можно запустить следующим образом:
 *   node charfreq.js < corpus.txt
 */

// Этот класс расширяет класс Map так, что метод get() возвращает вместо
// null указанное значение, когда ключ отсутствует в отображении.
class DefaultMap extends Map {
   constructor(defaultValue) {
     super();                              // Вызвать конструктор суперкласса
     this.defaultValue = defaultValue;  // Запомнить стандартное значение
   }

   get(key) {
     if (this.has(key)) {     //Если ключ присутствует в отображении,
       return super.get(key); //тогда возвратить его значение из суперкласса
     }
     else {
       return this.defaultValue;    //Иначе возвратить стандартное значение
     }
   }
}

//Этот класс рассчитывает и выводит гистограмму частоты использования букв.
class Histogram {
   constructor() {
     this.letterCounts = new DefaultMap(0);  //Отображение букв на счетчики
     this.totalLetters = 0;                  // Общее количество букв
   }
```

```
  // Эта функция обновляет гистограмму буквами текста.
  add(text) {
    // Удалить из текста пробельные символы
    // и преобразовать оставшиеся в верхний регистр
    text = text.replace(/\s/g, "").toUpperCase();

    // Пройти в цикле по символам текста
    for(let character of text) {
      let count = this.letterCounts.get(character);
                                                  // Получить старый счетчик
      this.letterCounts.set(character, count+1);
                                                  // Инкрементировать его
      this.totalLetters++;
    }
  }

  // Преобразовать гистограмму в строку, которая отображает графику ASCII
  toString() {
    // Преобразовать Map в массив массивов [ключ, значение]
    let entries = [...this.letterCounts];

    // Отсортировать массив по счетчикам, а затем в алфавитном порядке
    entries.sort((a,b) => {          // Функция для определения
                                     // порядка сортировки.
      if (a[1] === b[1]) {           // Если счетчики одинаковые, тогда
        return a[0] < b[0] ? -1 : 1; //сортировать в алфавитном порядке.
      } else {                       // Если счетчики отличаются, тогда
        return b[1] - a[1];          // сортировать по наибольшему счетчику.
      }
    });

    // Преобразовать счетчики в процентные отношения
    for(let entry of entries) {
      entry[1] = entry[1] / this.totalLetters*100;
    }

    // Отбросить все записи с процентным отношением менее 1%
    entries = entries.filter(entry => entry[1] >= 1);

    // Преобразовать каждую запись в строку текста
    let lines = entries.map(
      ([l,n]) => `${l}: ${"#".repeat(Math.round(n))} ${n.toFixed(2)}%`
    );

    // Возвратить сцепленные строки, разделенные символами новой строки
    return lines.join("\n");
  }


/ Эта асинхронная (возвращающая Promise) функция создает объект Histogram,
/ читает асинхронным образом порции текста из стандартного ввода
/ и добавляет их к гистограмме.
/ Достигнув конца потока данных, функция возвращает итоговую гистограмму.
```

```
async function histogramFromStdin() {
   process.stdin.setEncoding("utf-8");  // Читать строки Unicode, не байты
   let histogram = new Histogram();
   for await (let chunk of process.stdin) {
      histogram.add(chunk);
   }
   return histogram;
}

// Эта финальная строка кода является главным телом программы.
// Она создает объект Histogram из стандартного ввода
// и затем выводит гистограмму.
histogramFromStdin().then(histogram => { console.log(histogram.toString()); });
```

1.5. Резюме

В настоящей книге язык JavaScript объясняется с самого начала. Это значит, что мы начинаем с низкоуровневых деталей вроде комментариев, идентификаторов, переменных и типов, затем переходим к построению выражений, операторов, объектов и функций и далее раскрываем высокоуровневые языковые абстракции наподобие классов и модулей. Я серьезно отношусь к наличию в названии книги слов "подробное руководство", и в предстоящих главах язык будет объясняться с такой степенью детализации, которая поначалу может выглядеть обескураживающей. Однако настоящее владение языком JavaScript требует понимания деталей, и я надеюсь, что вы найдете время, чтобы как следует проработать эту книгу от корки до корки. Но, пожалуйста, не думайте, что вы обязаны поступать так при первом ее чтении. Если вы обнаруживаете, что застряли в каком-то разделе, тогда просто пропустите его. Получив практическое знание языка в целом, вы всегда можете вернуться обратно и отточить детали.

ГЛАВА 2

Лексическая структура

Лексическая структура языка программирования представляет собой набор элементарных правил, которые регламентируют написание программ на этом языке. Она отражает низкоуровневый синтаксис языка: указывает, каким образом должны выглядеть имена переменных, определяет символы разделителей для комментариев и устанавливает, как один оператор программы отделяется от другого, например. В текущей короткой главе документируется лексическая структура JavaScript. Здесь раскрываются следующие темы:

- чувствительность к регистру, пробелы и разрывы строк;
- комментарии;
- литералы;
- идентификаторы и зарезервированные слова;
- Unicode;
- необязательные точки с запятой.

2.1. Текст программы JavaScript

JavaScript — язык, чувствительный к регистру символов. Это значит, что ключевые слова языка, переменные, имена функций и другие *идентификаторы* всегда должны набираться с единообразным применением заглавных букв. Скажем, ключевое слово `while` нужно набирать как "while", но не "While" или "WHILE". Аналогично `online`, `Online`, `OnLine` и `ONLINE` — четыре отличающихся имени переменных.

Интерпретатор JavaScript игнорирует пробелы, встречающиеся между лексемами в программах. По большей части интерпретатор JavaScript также игнорирует разрывы строк (исключение описано в разделе 2.6). Благодаря возможности использования в программах пробелов и символов новой строки вы можете аккуратно и согласованно форматировать и делать отступы в коде, придавая ему легкий для чтения и понимания вид.

Вдобавок к обычному символу пробела (\u0020) интерпретатор JavaScript также распознает табуляции, разнообразные управляющие символы ASCII и различные символы пробелов Unicode как пробельные символы. Кроме того, интерпретатор JavaScript распознает символы новой строки, возврата каретки и последовательность возврат каретки/перевод строки как признаки конца строки.

2.2. Комментарии

В JavaScript поддерживаются два стиля комментариев. Любой текст после // и до конца строки трактуется как комментарий, который интерпретатор JavaScript игнорирует. Любой текст между символами /* и */ также рассматривается как комментарий; такие комментарии могут занимать несколько строк, но не могут быть вложенными. Ниже приведены все допустимые комментарии JavaScript:

```
// Это однострочный комментарий.

/* Это тоже комментарий */  // и еще один комментарий.

/*
 * Это многострочный комментарий. Добавочные символы * в начале каждой
 * строки не являются обязательной частью синтаксиса;
 * просто они классно выглядят!
 */
```

2.3. Литералы

Литерал — это значение данных, находящееся прямо в программе. Все нижеперечисленное представляет собой литералы:

```
12                  // Число двенадцать
1.2                 // Число одна целая и две десятых
"hello world"       // Строка текста
'Hi'                // Еще одна строка
true                // Булевское значение
false               // Другое булевское значение
null                // Отсутствие объекта
```

Исчерпывающие детали о числовых и строковых литералах ищите в главе 3.

2.4. Идентификаторы и зарезервированные слова

Идентификатор — это просто имя. Идентификаторы в JavaScript применяются для именования констант, переменных, свойств, функций и классов, а также для того, чтобы снабдить метками некоторые циклы в коде JavaScript. Идентификатор JavaScript должен начинаться с буквы, подчеркивания (_) или знака доллара ($). Последующими символами могут быть буквы, цифры, подчеркивания или знаки доллара. (Цифры не допускаются в качестве первого сим-

вола, а потому в JavaScript легко проводить различие между идентификаторами и числами.) Все приведенные далее идентификаторы являются допустимыми:

```
i
my_variable_name
v13
_dummy
$str
```

Подобно любому языку определенные идентификаторы в JavaScript зарезервированы для использования самим языком. Такие "зарезервированные слова" нельзя применять для обычных идентификаторов. Они перечислены в следующем разделе.

2.4.1. Зарезервированные слова

Перечисленные ниже слова являются частью самого языка JavaScript. Многие из них (вроде `if`, `while` и `for`) представляют собой зарезервированные слова, которые не должны использоваться для имен констант, переменных, функций или классов (хотя их можно применять для имен свойств внутри объекта). Другие (такие как `from`, `of`, `get` и `set`) используются в ограниченных контекстах без какой-либо синтаксической двусмысленности и совершенно законны в качестве идентификаторов. Третьи ключевые слова (наподобие `let`) не могут быть полностью зарезервированными по причине предохранения обратной совместимости с более старыми программами, поэтому существуют сложные правила, которые управляют тем, когда их можно применять как идентификаторы, а когда нет. (Например, `let` можно использовать для имени переменной в случае объявления с помощью `var` вне класса, но не внутри класса или с `const`.) Проще всего избегать применения любого из упомянутых слов в качестве идентификаторов кроме `from`, `set` и `target`, которые использовать безопасно и они уже часто употребляются.

```
as        const     export    get       null      target    void
async     continue  extends   if        of        this      while
await     debugger  false     import    return    throw     with
break     default   finally   in        set       true      yield
case      delete    for       instanceof static    try
catch     do        from      let       super     typeof
class     else      function  new       switch    var
```

В JavaScript также зарезервированы или ограничены в применении определенные ключевые слова, которые в текущее время не используются языком, но могут быть задействованы в будущих версиях:

```
enum implements interface package private protected public
```

По историческим причинам `arguments` и `eval` не разрешено применять для идентификаторов в некоторых обстоятельствах и лучше всего их вообще избегать в такой роли.

2.5. Unicode

Программы JavaScript пишутся с использованием набора символов Unicode и вы можете применять любые символы Unicode в строках и комментариях. Для переносимости и легкости редактирования в идентификаторах принято использовать только буквы ASCII и цифры. Но это лишь соглашение в программировании, а сам язык допускает наличие в идентификаторах букв, цифр и идеограмм Unicode (но не эмотиконов). Таким образом, программисты могут применять в качестве констант и переменных математические символы и неанглоязычные слова:

```
const π = 3.14;
const sí = true;
```

2.5.1. Управляющие последовательности Unicode

Аппаратное и программное обеспечение некоторых компьютеров не может отображать, вводить или корректно обрабатывать полный набор символов Unicode. Для поддержки программистов и систем, использующих более старые технологии, в JavaScript определены управляющие последовательности, которые позволяют записывать символы Unicode с применением только символов ASCII. Такие управляющие последовательности Unicode начинаются с символов \u, за которыми следуют либо четыре шестнадцатеричных цифры (буквы A–F верхнего или нижнего регистра), либо от одной до шести шестнадцатеричных цифр, заключенных в фигурные скобки. Управляющие последовательности Unicode могут присутствовать в строковых литералах JavaScript, литералах регулярных выражений и идентификаторах (но не в ключевых словах языка). Например, управляющая последовательность Unicode для символа "é" выглядит как \u00E9; существует три способа записи имени переменной, включающего указанный символ:

```
let café = 1; // Определение переменной с использованием символа Unicode
caf\u00e9     // => 1; доступ к переменной с применением
              // управляющей последовательности
caf\u{E9}     // => 1; еще одна форма той же управляющей
              // последовательности
```

В ранних версиях JavaScript поддерживались только управляющие последовательности с четырьмя цифрами. Версия с фигурными скобками была введена в ES6 для лучшей поддержки кодовых точек Unicode, которые требуют более 16 бит, таких как эмотиконы:

```
console.log("\u{1F600}"); // Выводит эмотикон с улыбающейся физиономией
```

Управляющие последовательности Unicode могут также находиться в комментариях, но поскольку комментарии игнорируются, в этом контексте они просто трактуются как символы ASCII и не интерпретируются как Unicode.

2.5.2. Нормализация Unicode

Если вы применяете в своих программах символы, отличающиеся от ASCII, то должны знать, что Unicode допускает более одного способа кодирования отдельного символа. Скажем, строку "é" можно кодировать как одиночный Unicode-символ \u00E9 или как обычный ASCII-символ "e", который комбинируется со знаком ударения \u0301. При отображении в текстовом редакторе такие две кодировки обычно выглядят одинаково, но они имеют отличающиеся двоичные кодировки, т.е. в JavaScript считаются разными, что может приводить к чрезвычайно запутанным программам:

```
const café = 1;  // Эта константа имеет имя "caf\u{e9}"
const café = 2;  // Это другая константа: "cafe\u{301}"
café  // => 1: эта константа имеет одно значение
café  // => 2: эта с виду неотличимая константа имеет другое значение
```

Стандарт Unicode определяет предпочтительную кодировку для всех символов и устанавливает процедуру нормализации, чтобы преобразовывать текст в каноническую форму, подходящую для сравнений. В JavaScript предполагается, что интерпретируемый исходный код уже нормализован и собственная нормализация *не* делается. Если вы планируете использовать символы Unicode в своих программах JavaScript, тогда должны удостовериться, что ваш редактор или какой-то другой инструмент выполняет нормализацию Unicode исходного кода, избегая появления разных, но визуально неотличимых идентификаторов.

2.6. Необязательные точки с запятой

Подобно многим языкам программирования точка с запятой (`;`) в JavaScript применяется для отделения операторов (см. главу 5) друг от друга. Это важно для прояснения смысла вашего кода: без разделителя конец одного оператора может показаться началом следующего оператора или наоборот. В JavaScript вы всегда можете опускать точку с запятой между двумя операторами, если они записаны в отдельных строках. (Вы также можете опустить точку с запятой в конце программы или если следующей лексемой в программе является закрывающая фигурная скобка: `}`.) Многие программисты на JavaScript (и код в настоящей книге) используют точки с запятой для явной пометки концов операторов, даже когда они не требуются. Еще один стиль предусматривает опускание точек с запятой всякий раз, когда это возможно, применяя их только в немногочисленных ситуациях, где они обязательны. Какой бы стиль вы ни выбрали, есть несколько деталей, которые вы должны знать о необязательных точках с запятой в JavaScript.

Рассмотрим показанный ниже код. Поскольку два оператора находятся в отдельных строках, первую точку с запятой можно было бы опустить:

```
a = 3;
b = 4;
```

Однако при следующей записи первая точка с запятой обязательна:

```
a = 3; b = 4;
```

Обратите внимание, что интерпретатор JavaScript не трактует каждый разрыв строки как точку с запятой: разрывы строк обычно трактуются как точки с запятой, только если синтаксический разбор кода невозможен без добавления неявной точки с запятой. Более формально (и с тремя исключениями, описанными чуть позже) интерпретатор JavaScript трактует разрыв строки как точку с запятой, если следующий непробельный символ не может быть интерпретирован как продолжение текущего оператора. Возьмем такой код:

```
let a
a
=
3
console.log(a)
```

Интерпретатор JavaScript воспринимает его подобно следующему коду:

```
let a; a = 3; console.log(a);
```

Интерпретатор JavaScript трактует первый разрыв строки как точку с запятой, потому что не может провести синтаксический разбор кода `let a a` без точки с запятой. Вторая переменная `a` могла бы остаться в одиночестве в виде оператора `a;`, но интерпретатор JavaScript не трактует второй разрыв строки как точку с запятой, поскольку может продолжить синтаксический разбор более длинного оператора `a = 3;`.

Указанные правила окончания операторов приводят к ряду неожиданных случаев. Приведенный далее код похож на два отдельных оператора, разделенных символом новой строки:

```
let y = x + f
(a+b).toString()
```

Но круглые скобки во второй строке кода могут интерпретироваться как вызов функции `f` из первой строки, и интерпретатор JavaScript трактует код следующим образом:

```
let y = x + f(a+b).toString();
```

Скорее всего, это не та интерпретация, которая задумывалась автором кода. Для работы кода как двух отдельных операторов требуется явная точка с запятой.

В общем случае, если оператор начинается с `(`, `[`, `/`, `+` или `-`, то есть шанс, что он может интерпретироваться как продолжение предыдущего оператора. Операторы, начинающиеся с `/`, `+` и `-`, на практике встречаются довольно редко, но операторы, начинающиеся с `(` и `[` — совсем не редкость, по крайней мере, в нескольких стилях программирования на JavaScript. Некоторым программистам нравится помещать защитную точку с запятой в начале любого такого оператора, чтобы он продолжал корректно работать, даже если предыдущий оператор модифицируется и ранее завершающая точка с запятой удаляется:

```
let x = 0                                 // Точка с запятой здесь опущена
;[x,x+1,x+2].forEach(console.log)         // Защитная ; сохраняет этот
                                          // оператор отдельным
```

Есть три исключения из общего правила, которому интерпретатор JavaScript следует при трактовке разрывов строк, когда синтаксический разбор второй строки как продолжения оператора в первой строке невозможен. Первое исключение касается операторов `return`, `throw`, `yield`, `break` и `continue` (см. главу 5). Перечисленные операторы часто оставляют в одиночестве, но временами за ними следует идентификатор или выражение. Если разрыв строки находится после любого их этих слов (перед любой другой лексемой), то интерпретатор JavaScript будет всегда воспринимать данный разрыв строки как точку с запятой. Например, если вы запишете:

```
return
true;
```

тогда интерпретатор JavaScript предполагает, что вы подразумеваете:

```
return; true;
```

Тем не менее, возможно, вы имели в виду:

```
return true;
```

Это значит, что вы не должны вставлять разрыв строки между `return`, `break` или `continue` и выражением, следующим за ключевым словом. Если вы вставите разрыв строки, то ваш код, по всей видимости, потерпит отказ неочевидным образом, трудно поддающимся отладке.

Второе исключение затрагивает операции ++ и -- (см. раздел 4.8). Указанные операции могут быть префиксными, находящимися перед выражением, или постфиксными, которые располагаются после выражения. Если вы хотите использовать любую из этих операций как постфиксную, тогда она должна находиться в той же строке, что и выражение, к которой операция применяется. Третье исключение связано с функциями, определенными с использованием "стрелочного" синтаксиса: сама стрелка => должна располагаться в той же строке, где находится список параметров.

2.7. Резюме

В главе было показано, как записываются программы JavaScript на самом низком уровне. В следующей главе мы поднимемся на одну ступеньку выше, представив элементарные типы и значения (числа, строки и т.д.), которые служат базовыми единицами вычислений для программ JavaScript.

ГЛАВА 3

Типы, значения и переменные

Компьютерные программы работают путем манипулирования значениями, такими как число 3.14 или текст "Hello World". Виды значений, которые можно представить и манипулировать ими в языке программирования, известны как типы, и одной из наиболее фундаментальных характеристик языка программирования является набор поддерживаемых типов. Когда программе необходимо запомнить значение с целью использования в будущем, она присваивает значение переменной (или "сохраняет" его в ней). Переменные имеют имена, которые можно употреблять в программах для ссылки на значения. Способ работы с переменными — еще одна фундаментальная характеристика любого языка программирования. В настоящей главе объясняются типы, значения и переменные в JavaScript. Она начинается с обзора и нескольких определений.

3.1. Обзор и определения

Типы JavaScript можно разделить на две категории: *элементарные типы* и *объектные типы*. Элементарные типы JavaScript включают числа, строки текста (называемые просто строками) и булевские истинностные значения (называемые просто булевскими). Существенная порция этой главы выделена под детальное объяснение числовых (см. раздел 3.2) и строковых (см. раздел 3.3) типов в JavaScript. Булевские значения раскрываются в разделе 3.4.

Специальные величины `null` и `undefined` в JavaScript относятся к элементарным значениям, но не являются числами, строками или булевскими значениями. Каждое значение обычно считается единственным членом собственного особого типа. Дополнительные сведения о `null` и `undefined` приведены в разделе 3.5. В ES6 добавлен новый специализированный тип, известный как символ (`Symbol`), который делает возможным определение языковых расширений, не причиняя вреда обратной совместимости. Значения `Symbol` кратко обсуждаются в разделе 3.6.

Любое значение JavaScript, которое отличается от числа, строки, булевского значения, символа, `null` или `undefined`, представляет собой объект. Объект

(т.е. член типа `Object`) — это коллекция *свойств*, где каждое свойство имеет имя и значение (либо элементарное значение, либо другой объект). В разделе 3.7 рассматривается один очень особый объект, *глобальный объект*, но более подробно объекты раскрываются в главе 6.

Обыкновенный объект JavaScript является неупорядоченной коллекцией именованных значений. В языке также определен специальный вид объекта, называемый массивом, который представляет упорядоченную коллекцию пронумерованных значений. Язык JavaScript включает специальный синтаксис для работы с массивами, а массивы обладают особым поведением, которое отличает их от обыкновенных объектов. Массивы обсуждаются в главе 7.

Вдобавок к базовым объектам и массивам в JavaScript определено несколько других полезных объектных типов. Объект `Set` представляет множество значений. Объект `Map` представляет отображение ключей на значения. Разнообразные типы "типизированных массивов" облегчают операции на массивах байтов и других двоичных данных. Тип `RegExp` представляет текстовые шаблоны и делает возможными сложно устроенные операции сопоставления, поиска и замены на строках. Тип `Date` представляет дату и время плюс поддерживает элементарную арифметику с датами. Тип `Error` и его подтипы представляют ошибки, которые могут возникать при выполнении кода JavaScript. Все упомянутые типы раскрываются в главе 11.

JavaScript отличается от более статичных языков тем, что функции и классы — не просто часть синтаксиса языка: они сами являются значениями, которыми можно манипулировать в программах JavaScript. Подобно любому не элементарному значению JavaScript функции и классы относятся к специализированному виду объектов. Они подробно рассматриваются в главах 8 и 9.

Интерпретатор JavaScript выполняет автоматическую сборку мусора для управления памятью. Это означает, что программист на JavaScript в основном не должен волноваться об уничтожении или освобождении объектов либо других значений. Когда значение больше не достижимо, т.е. программа не располагает ни одним способом ссылки на него, интерпретатору известно, что значение никогда не будет применяться снова, и он автоматически возвращает обратно занимаемую им память. (Программистам на JavaScript иногда нужно заботиться о том, чтобы по невнимательности значения не оставались достижимыми и не подлежащими освобождению дольше, нежели необходимо.)

Язык JavaScript поддерживает стиль объектно-ориентированного программирования. В широком смысле это значит, что вместо наличия глобально определенных функций для оперирования значениями различных типов типы самостоятельно определяют методы для работы со значениями. Скажем, чтобы отсортировать элементы массива `a`, мы не передаем `a` в функцию `sort()`. Взамен мы вызываем метод `sort()` массива `a`:

```
a.sort();    // Объектно-ориентированная версия sort(a).
```

Определение методов обсуждается в главе 9. Формально методы имеют только объекты JavaScript. Но числовые, строковые, булевские и символьные значения ведут себя так, будто они располагают методами. В JavaScript нельзя вызывать методы лишь на значениях `null` и `undefined`.

Объектные типы JavaScript являются *изменяемыми*, элементарные типы — *неизменяемыми*. Значение изменяемого типа можно менять: программа JavaScript способна изменять значения свойств объектов и элементов массивов. Числа, булевские значения, символы, `null` и `undefined` неизменяемы — например, не имеет смысла даже говорить о смене значения какого-то числа. Строки можно рассматривать как массивы символов и ожидать от них изменяемости. Однако строки в JavaScript неизменяемы: вы можете получать доступ к тексту по любому индексу строки, но нет какого-либо способа модификации текста существующей строки. Исследование отличий между изменяемыми и неизменяемыми значениями будет продолжено в разделе 3.8.

JavaScript свободно преобразует значения из одного типа в другой. Скажем, если программа ожидает строку, а вы предоставляете ей число, то произойдет автоматическое преобразование числа в строку. И если вы используете не булевское значение там, где ожидается булевское, тогда JavaScript соответствующим образом преобразует его. Правила преобразования значений объясняются в разделе 3.9. Либеральные правила преобразования значений JavaScript влияют на определение равенства, и операция равенства `==` выполняет преобразования типов, как описано в подразделе 3.9.1. (Тем не менее, на практике вместо операции равенства `==` рекомендуется применять операцию строгого равенства `===`, которая не делает никаких преобразований типов. Дополнительные сведения об обеих операциях ищите в подразделе 4.9.1.)

Константы и переменные позволяют использовать имена для ссылки на значения в программах. Константы объявляются с помощью `const`, а переменные посредством `let` (или `var` в более старом коде JavaScript). Константы и переменные JavaScript являются *нетипизированными*: в объявлениях не указывается, какой вид значений будет присваиваться. Объявление и присваивание переменных рассматривается в разделе 3.10.

Как должно быть ясно из длинного введения, это обширная глава, в которой объясняются многие фундаментальные аспекты, касающиеся представления и манипулирования данными в JavaScript. Мы начнем с погружения в детали чисел и текста JavaScript.

3.2. Числа

Основной числовой тип JavaScript, `Number`, служит для представления целых чисел и аппроксимации вещественных чисел. Числа в JavaScript представляются с применением 64-битного формата с плавающей точкой, определенного стандартом IEEE 754,[1] т.е. он способен представлять числа в диапазоне от $\pm 5 \times 10^{-324}$ до $\pm 1.7976931348623157 \times 10^{308}$.

Числовой формат JavaScript позволяет точно представлять все целые числа между –9 007 199 254 740 992 (-2^{53}) и 9 007 199 254 740 992 (2^{53}) включительно. В случае использования целочисленных значений, превышающих указанный

[1] Формат для чисел типа `double` в Java, C++ и большинстве современных языков программирования.

верхний предел, может теряться точность в хвостовых цифрах. Однако имейте в виду, что определенные операции в JavaScript (такие как индексация массивов и побитовые операции, описанные в главе 4) выполняются с 32-битными целыми числами. Если вам необходимо точно представлять более крупные целые числа, тогда обратитесь в подраздел 3.2.5.

Когда число находится прямо в программе JavaScript, оно называется *числовым литералом*. JavaScript поддерживает числовые литералы в нескольких форматах, которые будут описаны в последующих разделах. Обратите внимание, что любой числовой литерал может предваряться знаком минус (-), чтобы делать число отрицательным.

3.2.1. Целочисленные литералы

В программе JavaScript целое десятичное число записывается как последовательность цифр, например:

```
0
3
10000000
```

Помимо десятичных целочисленных литералов JavaScript распознает шестнадцатеричные (с основанием 16) значения. Шестнадцатеричный литерал начинается с символов 0x или 0X, за которыми следует строка шестнадцатеричных цифр. Шестнадцатеричная цифра — это одна из цифр от 0 до 9 или одна из букв от a (или A) до f (или F), которые представляют значения от 10 до 15. Вот примеры шестнадцатеричных целочисленных литералов:

```
0xff            // => 255: (15*16 + 15)
0xBADCAFE       // => 195939070
```

В ES6 и последующих версиях целые числа можно также выражать в двоичном (с основанием 2) или восьмеричном (с основанием 8) виде с применением префиксов 0b и 0o (или 0B и 0O) вместо 0x:

```
0b10101         // => 21:  (1*16 + 0*8 + 1*4 + 0*2 + 1*1)
0o377           // => 255: (3*64 + 7*8 + 7*1)
```

3.2.2. Числовые литералы с плавающей точкой

Числовые литералы с плавающей точкой могут содержать десятичную точку; они используют традиционный синтаксис для вещественных чисел. Вещественное значение представляется как целая часть числа, за которой следует десятичная точка и дробная часть числа.

Числовые литералы с плавающей точкой также могут быть представлены в экспоненциальной записи: вещественное число, за которым следует буква e (или E), необязательный знак плюс или минус и целочисленный показатель степени (экспонента). Такая запись представляет вещественное число, умноженное на 10 в степени экспоненты.

Более кратко синтаксис выглядит так:

[*цифры*] [. *цифры*] [(E|e) [(+|-)] *цифры*]

Например:

```
3.14
2345.6789
.333333333333333333
6.02e23             // 6.02 × 10^23
1.4738223E-32       // 1.4738223 × 10^-32
```

Разделители в числовых литералах

В длинных числовых литералах можно применять подчеркивания, чтобы разбивать их на порции, которые легче для восприятия:

```
let billion = 1_000_000_000;  // Подчеркивание как разделитель тысяч.
let bytes = 0x89_AB_CD_EF;    // Подчеркивание как разделитель байтов.
let bits = 0b0001_1101_0111;  // Подчеркивание как разделитель полубайтов.
let fraction = 0.123_456_789; // Работает и в дробной части.
```

На момент написания главы в начале 2020 года подчеркивания в числовых литералах еще не были формально стандартизированы в качестве части JavaScript. Но они находятся на поздней фазе процесса стандартизации и внедрены во всех основных браузерах и Node.

3.2.3. Арифметические действия в JavaScript

Программы JavaScript работают с числами, используя арифметические операции, которые предоставляет язык. К ним относятся + для сложения, – для вычитания, * для умножения, / для деления и % для деления по модулю (получения остатка от деления). В ES2016 добавляется ** для возведения в степень. Полные сведения об указанных и других операциях ищите в главе 4.

В дополнение к таким базовым арифметическим операциям JavaScript поддерживает более сложные математические действия через набор функций и констант, определенных как свойства объекта Math:

```
Math.pow(2,53)      // => 9007199254740992: 2 в степени 53
Math.round(.6)      // => 1.0: округляет до ближайшего целого
Math.ceil(.6)       // => 1.0: округляет в большую сторону до целого
Math.floor(.6)      // => 0.0: округляет в меньшую сторону до целого
Math.abs(-5)        // => 5: абсолютная величина
Math.max(x,y,z)     // Возвращает наибольший аргумент
Math.min(x,y,z)     // Возвращает наименьший аргумент
Math.random()       // Псевдослучайное число x, где 0 <= x < 1.0
Math.PI             // π: длина окружности, деленная на диаметр
Math.E              // e: основание натурального логарифма
Math.sqrt(3)        // => 3**0.5: квадратный корень из 3
Math.pow(3, 1/3)    // => 3**(1/3): кубический корень из 3
Math.sin(0)         // Тригонометрия: также есть Math.cos, Math.atan и т.д.
Math.log(10)        // Натуральный логарифм 10
Math.log(100)/Math.LN10     // Десятичный логарифм 100
Math.log(512)/Math.LN2      // Двоичный логарифм 512
Math.exp(3)                 // Math.E в кубе
```

В ES6 определены дополнительные функции объекта Math:

```
Math.cbrt(27)       // => 3: кубический корень
Math.hypot(3, 4)    // => 5: квадратный корень из суммы квадратов
                    //       всех аргументов
Math.log10(100)     // => 2: десятичный логарифм
Math.log2(1024)     // => 10: двоичный логарифм
Math.log1p(x) // Натуральный логарифм (1+x); точен для очень малых x
Math.expm1(x) // Math.exp(x)-1; инверсия Math.log1p()
Math.sign(x)  // -1, 0 или 1 для аргументов <, == или > 0
Math.imul(2,3)// => 6: оптимизированное умножение 32-битных целых чисел
Math.clz32(0xf)     // => 28: количество ведущих нулевых бит
                    //        в 32-битном целом числе
Math.trunc(3.9)     // => 3: преобразует в целое число,
                    //       отбрасывая дробную часть
Math.fround(x)      // Округляет до ближайшего 32-битного числа
                    // с плавающей точкой
Math.sinh(x)        // Гиперболический синус.
                    // Также есть Math.cosh(), Math.tanh()
Math.asinh(x)       // Гиперболический арксинус.
                    // Также есть Math.acosh(), Math.atanh()
```

Арифметические действия в JavaScript не генерируют ошибки в случаях переполнения, потери значимости или деления на ноль. Когда результат выполнения числовой операции превышает наибольшее представимое число (переполнение), то результатом будет особое значение бесконечности, `Infinity`. Аналогично, когда абсолютная величина отрицательного значения превышает абсолютную величину наибольшего представимого отрицательного числа, то результатом будет отрицательная бесконечность, `-Infinity`. Поведение значений бесконечностей вполне ожидаемо: их сложение, вычитание, умножение или деление на что угодно дает в результате значение бесконечности (возможно с обратным знаком).

Потеря значимости возникает, когда результат числовой операции находится ближе к нулю, чем наименьшее представимое число. В таком случае JavaScript возвращает 0. Если потеря значимости происходит с отрицательным числом, тогда JavaScript возвращает специальное значение, известное как "отрицательный ноль". Это значение практически неотличимо от нормального нуля и программисты на JavaScript редко нуждаются в его выявлении.

Деление на ноль в JavaScript ошибкой не является: просто возвращается бесконечность или отрицательная бесконечность. Тем не менее, существует одно исключение: ноль, деленный на ноль, не имеет четко определенного значения, и результатом такой операции будет особое значение "не число" (not-a-number), `NaN`. Кроме того, `NaN` также возникает при попытке деления бесконечности на бесконечность, взятия квадратного корня из отрицательного числа или применения арифметических операций с нечисловыми операндами, которые не могут быть преобразованы в числа.

Для хранения положительной бесконечности и значения "не число" в JavaScript предопределены глобальные константы Infinity и NaN, значения которых также доступны в виде свойств объекта Number:

```
Infinity                    // Положительное число, слишком большое
                            // для представления
Number.POSITIVE_INFINITY    // То же значение
1/0                         // => Infinity
Number.MAX_VALUE * 2        // => Infinity; переполнение
-Infinity                   // Отрицательное число, слишком большое
                            // для представления
Number.NEGATIVE_INFINITY    // То же значение
-1/0                        // => -Infinity
-Number.MAX_VALUE * 2       // => -Infinity

NaN                         // Значение "не число"
Number.NaN                  // То же значение, записанное по-другому
0/0                         // => NaN
Infinity/Infinity           // => NaN

Number.MIN_VALUE/2          // => 0: потеря значимости
-Number.MIN_VALUE/2         // => -0: отрицательный ноль
-1/Infinity                 // => -0: тоже отрицательный 0
-0

// Следующие свойства Number определяются в ES6
Number.parseInt()       // То же, что и глобальная функция parseInt()
Number.parseFloat()     // То же, что и глобальная функция parseFloat()
Number.isNaN(x)         // Является ли x значением NaN?
Number.isFinite(x)      // Является ли x конечным числом?
Number.isInteger(x)     // Является ли x целым числом?
Number.isSafeInteger(x) // Является ли x целым числом -(2**53) < x < 2**53?
Number.MIN_SAFE_INTEGER // => -(2**53 - 1)
Number.MAX_SAFE_INTEGER // => 2**53 - 1
Number.EPSILON          // => 2**-52: наименьшая разница между числами
```

Значение "не число" в JavaScript обладает одной необычной особенностью: оно не равно никакому другому значению, включая самого себя. Отсюда следует, что вы не можете записать x === NaN, чтобы выяснить, имеет ли переменная x значение NaN. Взамен вы должны записывать x != x или Number.isNaN(x). Указанные выражения будут истинными тогда и только тогда, когда x содержит то же самое значение, что и глобальная константа NaN.

Глобальная функция isNaN() аналогична Number.isNaN(). Она возвращает true, если аргументом является NaN или аргумент имеет нечисловое значение, которое не может быть преобразовано в число. Связанная функция Number.isFinite() возвращает true, если аргумент представляет собой число, отличающееся от NaN, Infinity или -Infinity. Глобальная функция isFinite() возвращает true, если ее аргумент является или может быть преобразован в конечное число.

Значение отрицательного нуля тоже кое в чем необычно. Оно равно (даже при использовании проверки на предмет строгого равенства JavaScript) положительному нулю, т.е. два значения почти неразличимы кроме ситуации, когда они применяются в качестве делителя:

```
let zero = 0;          // Нормальный ноль
let negz = -0;         // Отрицательный ноль
zero === negz          // => true: ноль и отрицательный ноль равны
1/zero === 1/negz      // => false: Infinity и -Infinity не равны
```

3.2.4. Двоичное представление чисел с плавающей точкой и ошибки округления

Вещественных чисел бесконечно много, но с помощью формата плавающей точки JavaScript может быть представлено только конечное их количество (точнее 18 437 736 874 454 810 627 чисел). Это означает, что при работе с вещественными числами в JavaScript представление числа часто будет аппроксимацией фактического числа.

Представление плавающей точки IEEE-754, используемое JavaScript (и почти любым другим современным языком программирования), является двоичным и может точно представлять дроби вроде `1/2`, `1/8` и `1/1024`. К сожалению, чаще всего (особенно при выполнении финансовых расчетов) мы применяем десятичные дроби: `1/10`, `1/100` и т.д. Двоичные представления чисел с плавающей точкой не способны точно представлять даже такие простые числа, как `0.1`.

Числа JavaScript обладают достаточной точностью и могут обеспечить очень близкую аппроксимацию к `0.1`. Но тот факт, что это число не может быть представлено точно, способен привести к проблемам. Взгляните на следующий код:

```
let x = .3 - .2;       // тридцать центов минус двадцать центов
let y = .2 - .1;       // двадцать центов минус десять центов
x === y                // => false: два значения не одинаковы!
x === .1               // => false: .3-.2 не равно .1
y === .1               // => true: .2-.1 равно .1
```

Из-за ошибки округления разница между аппроксимациями `.3` и `.2` оказывается не точно такой же, как разница между аппроксимациями `.2` и `.1`. Важно понимать, что продемонстрированная проблема не специфична для JavaScript: она затрагивает любой язык программирования, в котором используется двоичное представление чисел с плавающей точкой. Кроме того, имейте в виду, что значения `x` и `y` в показанном здесь коде очень близки друг к другу и к корректной величине. Вычисленные значения пригодны практически для любой цели; проблема возникает лишь при попытке сравнения значений на предмет равенства.

Если такие аппроксимации с плавающей точкой проблематичны для ваших программ, тогда обдумайте применение масштабированных целых чисел. Например, вы можете манипулировать денежными величинами как целочисленными центами, а не дробными долларами.

3.2.5. Целые числа произвольной точности с использованием BigInt

Одним из новейших средств JavaScript, определенных в ES2020, является числовой тип под названием BigInt. По состоянию на начало 2020 года он был внедрен в Chrome, Firefox, Edge и Node плюс готовилась реализация в Safari. Как следует из названия, BigInt — это числовой тип с целыми значениями. Тип BigInt был добавлен в JavaScript главным образом для того, чтобы сделать возможным представление 64-битных целых чисел, которое требуется для совместимости со многими другими языками программирования и API-интерфейсами. Но значения BigInt могут иметь тысячи и даже миллионы цифр, если вам действительно необходимо работать с настолько большими числами. (Однако обратите внимание, что реализации BigInt не подходят для криптографии, поскольку они не пытаются предотвратить атаки по времени.)

Литералы BigInt записываются как строка цифр, за которой следует буква n в нижнем регистре. По умолчанию они десятичные, но можно применять префиксы 0b, 0o и 0x для двоичных, восьмеричных и шестнадцатеричных BigInt:

```
1234n                          // Не настолько большой литерал BigInt
0b111111n                      // Двоичный литерал BigInt
0o7777n                        // Восьмеричный литерал BigInt
0x8000000000000000n            // => 2n**63n: 64-битное целое
```

Для преобразования обыкновенных чисел или строк JavaScript в значения BigInt можно использовать функцию BigInt():

```
BigInt(Number.MAX_SAFE_INTEGER)     // => 9007199254740991n
let string = "1" + "0".repeat(100); // 1 со следующими 100 нулями
BigInt(string)                      // => 10n**100n: один гугол
```

Арифметические действия со значениями BigInt работают подобно арифметическим действиям с обыкновенными числами JavaScript за исключением того, что деление отбрасывает любой остаток и округляет в меньшую сторону (по направлению к нулю):

```
1000n + 2000n                  // => 3000n
3000n - 2000n                  // => 1000n
2000n * 3000n                  // => 6000000n
3000n / 997n                   // => 3n: частное равно 3
3000n % 997n                   // => 9n: остаток равен 9
(2n ** 131071n) - 1n           // Простое число Мерсенна,
                               // имеющее 39457 десятичных цифр
```

Хотя стандартные операции +, -, *, /, % и ** работают с BigInt, важно понимать, что смешивать операнды типа BigInt с обычными числовыми операндами нельзя. Поначалу это может показаться странным, но для этого есть все основания. Если бы один числовой тип был более общим, чем другой, то определить арифметические действия на смешанных операндах было бы легче, просто возвращая значение более общего типа. Однако ни один из типов не является более общим, чем другой. Дело в том, что BigInt может представ-

лять чрезвычайно большие значения, поэтому его можно считать более общим, чем обычные числа. Но BigInt способен представлять только целые числа, что делает более общим обычный числовой тип JavaScript. Обойти указанную проблему невозможно, а потому в арифметических операциях JavaScript просто не разрешено смешивать операнды разных числовых типов.

Напротив, операции сравнения работают со смешанными числовыми типами (в подразделе 3.9.1 объясняется отличие между == и ===):

```
1 < 2n          // => true
2 > 1n          // => true
0 == 0n         // => true
0 === 0n        // => false: операция === также проверяет
                //           эквивалентность типов
```

Побитовые операции (описанные в подразделе 4.8.3) в целом работают с операндами BigInt. Тем не менее, ни одна из функций объекта Math не принимает операнды BigInt.

3.2.6. Дата и время

В JavaScript определен простой класс Date для представления и манипулирования числами, которые представляют дату и время. Экземпляры класса Date являются объектами, но они также имеют числовое представление в виде *отметок времени*, которые указывают количество миллисекунд, прошедших со дня 1 января 1970 года:

```
let timestamp = Date.now(); //Текущее время как отметка времени (число).
let now = new Date();       // Текущее время как объект Date.
let ms = now.getTime();     // Преобразовать в миллисекундную
                            // отметку времени.
let iso = now.toISOString(); // Преобразовать в строку со стандартным
                            // форматом.
```

Класс Date и его методы подробно рассматриваются в разделе 11.4. Но объекты Date снова встретятся в подразделе 3.9.3, где исследуются детали преобразований типов JavaScript.

3.3. Текст

В качестве типа JavaScript для представления текста применяется *строка*. Строка — это неизменяемая упорядоченная последовательность 16-битных значений, каждое из которых обычно представляет символ Unicode. *Длиной* строки является количество содержащихся в ней 16-битных значений. Строки (и массивы) в JavaScript используют индексацию, начинающуюся с нуля: первое 16-битное значение находится в позиции 0, второе — в позиции 1 и т.д. *Пустая строка* — это строка с длиной 0. В JavaScript не предусмотрен особый тип, который представлял бы одиночный элемент строки. Для представления одиночного 16-битного значения просто применяйте строку, имеющую длину 1.

JavaScript использует кодировку UTF-16 набора символов Unicode, а строки JavaScript являются последовательностями 16-битных значений без знака. Наиболее часто применяемые символы Unicode (из "основной многоязычной плоскости") имеют кодовые точки, которые умещаются в 16 бит и могут быть представлены одним элементом строки. Символы Unicode, кодовые точки которых не умещаются в 16 бит, кодируются с использованием правил UTF-16 в виде последовательностей (известных как "суррогатные пары") из двух 16-битных значений. Таким образом, строка JavaScript с длиной 2 (два 16-битных значения) вполне может представлять только один символ Unicode:

```
let euro = "€";
let love = "❤";
euro.length    // => 1: этот символ имеет один 16-битный элемент
love.length    // => 2: кодом UTF-16 символа ❤ является "\ud83d\udc99"
```

Большинство методов манипулирования строками, определенных в JavaScript, действуют на 16-битных значениях, а не на символах. Они не трактуют суррогатные пары особым образом, не выполняют нормализацию строки и даже не проверяют, правильно ли построена строка с точки зрения UTF-16.

Однако в ES6 строки являются *итерируемыми*, и применение цикла `for/of` или операции `...` со строкой приведет к проходу по действительным символам строки, а не по 16-битным значениям.

3.3.1. Строковые литералы

Чтобы включить строку в программу JavaScript, просто поместите символы строки внутрь совпадающей пары одинарных, двойных или обратных кавычек (', " или `). Символы двойных и обратных кавычек могут содержаться внутри строк, ограниченных символами одинарных кавычек, и то же самое справедливо для строк в двойных и обратных кавычках. Ниже показаны примеры строковых литералов:

```
""  // Пустая строка: содержит ноль символов
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"π is the ratio of a circle's circumference to its radius"
`"She said 'hi'", he said.`
```

Строки в обратных кавычках являются возможностью ES6 и допускают наличие в строковых литералах встроенных (или *интерполированных*) выражений JavaScript. Синтаксис интерполяции выражений обсуждается в подразделе 3.3.4.

Первоначальные версии JavaScript требовали, чтобы строковые литералы были записаны в одной строке, поэтому часто можно видеть код JavaScript, в котором длинные строки создаются за счет конкатенации однострочных строк с помощью операции +. Тем не менее, начиная с ES5, строковый литерал можно разбивать на множество строк, помещая в конец каждой строки кроме последней обратную косую черту (\). Ни обратная косая черта, ни находящийся за ней разделитель строк не входят в состав строкового литерала. Если вам нужно включить символ новой строки в строковый литерал, помещенный в одинарные или двойные кавычки, тогда используйте последовательность символов \n (объясняется в следующем разделе). Синтаксис с обратными кавычками, введенный в ES6, разрешает разбивать строки на множество строк, и в этом случае разделители строк будут частью строкового литерала:

```
// Строка, представляющая две строки, которые записаны в одной строке:
'two\nlines'

// Однострочная строка, записанная в трех строках:
"one\
 long\
 line"

// Двухстрочная строка, записанная в двух строках:
`the newline character at the end of this line
is included literally in this string`
```

Обратите внимание, что в случае применения одинарных кавычек для ограничения строк вы должны соблюдать осторожность при записи англоязычных сокращенных и притяжательных форм, таких как *can't* и *O'Reilly's*. Поскольку апостроф — то же самое, что и одинарная кавычка, вы обязаны использовать символ обратной косой черты (\) для "отмены действия" любых апострофов, присутствующих внутри строк в одинарных кавычках (управляющие последовательности рассматриваются в следующем разделе).

При программировании на стороне клиента код JavaScript может содержать строки кода HTML, а код HTML включать строки кода JavaScript. Как и JavaScript, для ограничения строк в HTML применяются либо одинарные, либо двойные кавычки. Таким образом, при комбинировании кода JavaScript и HTML имеет смысл использовать один стиль кавычек для JavaScript и другой стиль для HTML. В приведенном ниже примере строка "Thank you" заключена в одинарные кавычки внутри выражения JavaScript, которое помещено в двойные кавычки в атрибуте обработчика событий HTML:

```
<button onclick="alert('Thank you')">Click Me</button>
```

3.3.2. Управляющие последовательности в строковых литералах

Символ обратной косой черты (\) имеет специальное назначение в строках JavaScript. В сочетании со следующим за ним символом он представляет символ, который по-другому не может быть представлен внутри строки. Например, \n является *управляющей последовательностью*, которая представляет символ новой строки.

Еще одним примером, упомянутым ранее, служит последовательность \', которая представляет символ одинарной кавычки (или апострофа). Такая управляющая последовательность полезна, когда необходимо включить апостроф в строковый литерал, находящийся внутри одинарных кавычек. Вот почему они называются управляющими последовательностями: обратная косая черта позволяет управлять интерпретацией символа одинарной кавычки. Вместо его применения для пометки конца строки он используется в качестве апострофа:

```
'You\'re right, it can\'t be a quote'
```

В табл. 3.1 перечислены управляющие последовательности JavaScript и символы, которые они представляют. Три управляющих последовательности являются обобщенными и могут применяться для представления любого символа за счет указания его кода Unicode в виде шестнадцатеричного числа. Скажем, последовательность \xA9 представляет символ авторского права, который имеет код Unicode, заданный шестнадцатеричным числом A9. Аналогично последовательность \u представляет произвольный символ Unicode, указанный четырьмя шестнадцатеричными цифрами или от одной до шести цифр, когда они заключены в фигурные скобки: например, \u03c0 представляет символ π, а \u{1f600} — эмотикон "ухмыляющаяся рожица".

Таблица 3.1. Управляющие последовательности JavaScript

Последовательность	Представляемый символ
\0	Символ NUL (\u0000)
\b	Забой (\u0008)
\t	Горизонтальная табуляция (\u0009)
\n	Новая строка (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Апостроф или одинарная кавычка (\u0027)
\\	Обратная косая черта (\u005C)
\x*nn*	Символ Unicode, указанный двумя шестнадцатеричными цифрами *nn*
\u*nnnn*	Символ Unicode, указанный четырьмя шестнадцатеричными цифрами *nnnn*
\u{*n*}	Символ Unicode, указанный с помощью кодовой точки *n*, где n — от одной до шести шестнадцатеричных цифр между 0 и 10FFFF (ES6)

Если символ \ предшествует любому символу, не перечисленному в табл. 3.1, тогда обратная косая черта просто игнорируется (хотя в будущих версиях языка, конечно же, могут быть определены новые управляющие последовательности). Например, \# — то же самое, что и #. Наконец, как отмечалось ранее, ES5

разрешает использовать обратную косую черту перед разрывом строки, чтобы разбить строковый литерал на несколько строк.

3.3.3. Работа со строками

Одним из встроенных средств JavaScript является возможность *конкатенации* строк. Если операция + применяется с цифрами, то она складывает их. Но если операция + используется со строками, тогда она объединяет их, присоединяя вторую строку в конец первой. Вот пример:

```
let msg = "Hello, " + "world";    // Образует строку "Hello, world"
let greeting = "Welcome to my blog," + " " + name;
```

Строки можно сравнивать с помощью стандартных операций равенства === и неравенства !==: две строки равны, если и только если они содержат в точности одну и ту же последовательность 16-битных значений. Строки также можно сравнивать с применением операций <, <=, > и >=. Сравнение строк выполняется путем сравнения 16-битных значений. (В подразделе 11.7.3 описаны более надежные способы сравнения и сортировки строк, учитывающие локаль.)

Для определения длины строки — количества содержащихся в ней 16-битных значений — используется свойство `length` строки:

```
s.length
```

В дополнение к свойству `length` в JavaScript предлагается богатый API-интерфейс для работы со строками:

```
let s = "Hello, world";      // Начать с некоторого текста.

// Получение порций строки
s.substring(1,4)             // => "ell": 2-й, 3-й и 4-й символы
s.slice(1,4)                 // => "ell": то же самое
s.slice(-3)                  // => "rld": последние 3 символа
s.split(", ")                // => ["Hello", "world"]: разбивает по
                             // строке разделителя

// Поиск в строке
s.indexOf("l")               // => 2: позиция первой буквы l
s.indexOf("l", 3)            // => 3: позиция первой буквы l,
                             // начиная с 3-й позиции
s.indexOf("zz")              // => -1: s не включает подстроку "zz"
s.lastIndexOf("l")           // => 10: позиция последней буквы l

// Булевские функции поиска в ES6 и последующих версиях
s.startsWith("Hell")         // => true: строка начинается с этого
s.endsWith("!")              // => false: s не оканчивается этим
s.includes("or")             // => true: s включает подстроку "or"

// Создание модифицированных версий строки
s.replace("llo", "ya")       // => "Heya, world"
s.toLowerCase()              // => "hello, world"
s.toUpperCase()              // => "HELLO, WORLD"
s.normalize()  // Нормализация Unicode NFC (Normalization Form C): ES6
```

```
s.normalize("NFD") // Нормализация Unicode NFD (Normalization Form D).
                   // Также есть "NFKC", "NFKD"

// Инспектирование индивидуальных (16-битных) символов строки
s.charAt(0)        // => "H": первый символ
s.charAt(s.length-1)    // => "d": последний символ
s.charCodeAt(0)    // => 72: 16-битное число в указанной позиции
s.codePointAt(0)   // => 72: ES6, работает с кодовыми точками > 16 бит

// Функции дополнения строк в ES2017
"x".padStart(3)    // => " x": добавляет пробелы слева до длины 3
"x".padEnd(3)      // => "x ": добавляет пробелы справа до длины 3
"x".padStart(3,"*")// => "**x": добавляет звездочки слева до длины 3
"x".padEnd(3, "-") // => "x--": добавляет дефисы справа до длины 3

// Функции усечения пробелов. trim() введена в ES5; остальные в ES2019
" test ".trim()         // => "test": удаляет пробелы в начале и конце
" test ".trimStart()    // => "test ": удаляет пробелы слева.
                        // Также есть trimLeft
" test ".trimEnd()      // => " test": удаляет пробелы справа.
                        // Также есть trimRight

// Смешанные методы строк
s.concat("!")           // => "Hello, world!": взамен просто
                        // используйте операцию +
"<>".repeat(5)          // => "<><><><><>": выполняет конкатенацию
                        // n копий. ES6
```

Не забывайте, что строки в JavaScript неизменяемы. Методы вроде `replace()` и `toUpperCase()` возвращают новые строки: они не модифицируют строку, на которой вызываются.

Строки также можно трактовать как массивы, допускающие только чтение, и для получения доступа к индивидуальным символам (16-битным значениям) строки вместо вызова метода `charAt()` разрешено применять квадратные скобки:

```
let s = "hello, world";
s[0]                    // => "h"
s[s.length-1]           // => "d"
```

3.3.4. Шаблонные литералы

В ES6 и последующих версиях строковые литералы могут ограничиваться обратными кавычками:

```
let s = `hello world`;
```

Однако это больше, чем просто еще один синтаксис строковых литералов, поскольку такие *шаблонные литералы* (template literal) могут включать произвольные выражения JavaScript. Финальное значение строкового литерала в обратных кавычках вычисляется за счет оценки всех включенных выражений, преобразования значений этих выражений в строки и объединения вычисленных строк с литеральными символами внутри обратных кавычек:

```
let name = "Bill";
let greeting = `Hello ${ name }.`;  // greeting == "Hello Bill."
```

Все, что расположено между символами ${ и соответствующим символом }, интерпретируется как выражение JavaScript. Все, что находится вне фигурных скобок, является нормальным текстом строкового литерала. Выражение внутри скобок оценивается и затем преобразуется в строку, которая вставляется в шаблон, замещая знак доллара, фигурные скобки и все, что между ними.

Шаблонный литерал может включать любое количество выражений. В нем могут использоваться любые управляющие символы, допускающиеся в нормальных строках, и он может быть разнесен на любое количество строк без каких-либо специальных символов отмены действия. Следующий шаблонный литерал содержит четыре выражения JavaScript, управляющую последовательность Unicode и, по крайней мере, четыре символа новой строки (значения переменных также могут включать в себя символы новой строки):

```
let errorMessage = `\
\u2718 Test failure at ${filename}:${linenumber}:
${exception.message}
Stack trace:
${exception.stack}
`;
```

Обратная косая черта в конце первой строки отменяет действие начального конца строки, так что результирующая строка начинается с символа Unicode ✘ (\u2718), а не с новой строки.

Теговые шаблонные литералы

Мощная, но менее часто применяемая возможность шаблонных литералов, заключается в том, что если сразу после открывающей обратной кавычки находится имя функции (или "тег"), тогда текст и выражения внутри шаблонного литерала передаются этой функции. Значением такого "тегового шаблонного литерала" (tagged template literal) будет возвращаемое значение функции. Данную особенность можно использовать, скажем, для отмены действия определенных символов в значениях HTML или SQL перед их вставкой в текст.

В ES6 имеется одна встроенная теговая функция: `String.raw()`. Она возвращает текст внутри обратных кавычек, не обрабатывая управляющие символы в обратных кавычках:

```
`\n`.length    // => 1: строка содержит одиночный символ новой строки
String.raw`\n`.length  // => 2: строка содержит символ обратной косой
                       // черты и букву n
```

Обратите внимание, что хотя часть тегового шаблонного литерала, относящая к тегу, является функцией, круглые скобки в вызове отсутствуют. В показанном очень специфичном случае символы обратных кавычек заменяют открывающую и закрывающую круглые скобки.

Возможность определения собственных теговых функций шаблонов — мощное средство JavaScript. Такие функции не обязаны возвращать строки и их можно применять подобно конструкторам, как если бы они определяли новый литеральный синтаксис для языка. Пример будет демонстрироваться в разделе 14.5.

3.3.5. Сопоставление с шаблонами

В JavaScript определен тип данных, который называется *регулярным выражением* (regular expression, или RegExp) и предназначен для описания и сопоставления с шаблонами в строках текста. Регулярные выражения в JavaScript не относятся к одному из фундаментальных типов языка, но они имеют литеральный синтаксис как у чисел и строк, поэтому иногда кажутся фундаментальными. Грамматика литералов с регулярными выражениями является сложной, а определяемый ими API-интерфейс нетривиален. Регулярные выражения подробно документируются в разделе 11.3. Тем не менее, учитывая мощь регулярных выражений и тот факт, что они часто используются при обработке текста, в текущем разделе предлагается краткий образ.

Текст между парой обратных косых черт образует литерал регулярного выражения. Вторая обратная косая черта в паре также может сопровождаться одной и более букв, которые модифицируют смысл шаблона. Ниже приведены примеры:

```
/^HTML/;            // Соответствует буквам H T M L в начале строки
/[1-9][0-9]*/;      // Соответствует отличной от нуля цифре,
                    // за которой следует любое количество цифр
/\bjavascript\b/i; // Соответствует "javascript" как слову,
                    // нечувствительно к регистру символов
```

Объекты RegExp определяют ряд полезных методов, а строки имеют методы, которые принимают аргументы RegExp, например:

```
let text = "testing: 1, 2, 3";   // Пример текста
let pattern = /\d+/g;            // Соответствует всем вхождениям одной
                                 // или большего количества цифр

pattern.test(text)               // => true: есть совпадение
text.search(pattern)             // => 9: позиция первого совпадения
text.match(pattern)              // => ["1", "2", "3"]:
                                 //    массив всех совпадений
text.replace(pattern, "#")       // => "testing: #, #, #"
text.split(/\D+/)                // => ["","1","2","3"]: разбиение
                                 // по нецифровым символам
```

3.4. Булевские значения

Булевское значение представляет истинность или ложность, "включено" или "выключено", "да" или "нет". Существует только два возможных значения этого типа, которые представлены с помощью зарезервированных слов `true` и `false`.

Булевские значения обычно являются результатом сравнений, которые делаются в программах JavaScript, скажем:

```
a === 4
```

Данный код проверяет, равно ли значение переменной a числу 4. Если равно, тогда результатом сравнения будет булевское значение `true`. Если a не равно 4, то результатом сравнения будет `false`.

Булевские значения часто применяются в управляющих структурах JavaScript. Например, оператор `if/else` в JavaScript выполняет одно действие, если булевское значение равно `true`, и другое действие, если оно равно `false`. Сравнение, которое создает булевское значение, обычно объединяется прямо с использующим его оператором. Результат выглядит следующим образом:

```
if (a === 4) {
   b = b + 1;
} else {
   a = a + 1;
}
```

Код проверяет, равна ли переменная a числу 4. Если это так, тогда 1 прибавляется к b, а иначе к a.

Как обсуждалось в разделе 3.9, любое значение JavaScript может быть преобразовано в булевское значение. Показанные ниже значения преобразуются и потому работают подобно `false`:

```
undefined
null
0
-0
NaN
""  // пустая строка
```

Все остальные значения, включая все объекты (и массивы) преобразуются и потому работают как `true`. Временами значение `false` и шесть значений, которые в него преобразуются, называют *ложными*, тогда как все остальные значения — *истинными*. Каждый раз, когда JavaScript ожидает булевское значение, ложное значение работает подобно `false`, а истинное — подобно `true`.

В качестве примера предположим, что переменная o хранит либо какой-то объект, либо значение `null`. С помощью следующего оператора `if` можно выполнить явную проверку, отличается ли o от `null`:

```
if (o !== null) ...
```

Операция неравенства `!==` сравнивает o с `null` и в результате дает либо `true`, либо `false`. Но сравнение можно опустить и взамен полагаться на тот факт, что `null` является ложным, а объекты — истинными:

```
if (o) ...
```

В первом случае тело оператора `if` будет выполняться, только если o не равно `null`. Второй случай менее строгий: тело `if` будет выполняться, только если o не равно `false` или любому ложному значению (такому как `null` или `undefined`). Какой оператор `if` будет более подходящим для вашей программы, в действительности зависит от того, что за значения вы рассчитываете присваивать o. Если вам необходимо различать `null` от 0 и `""`, тогда вы должны применять явное сравнение.

Булевские значения имеют метод `toString()`, который можно использовать для их преобразования в строки `"true"` или `"false"`, но какими-то дру-

гими полезными методами они не обладают. Несмотря на тривиальный API-интерфейс, есть три важных булевских операции.

Операция `&&` выполняет булевскую операцию "И". Она дает истинное значение тогда и только тогда, когда оба ее операнда истинны, а в противном случае оценивается в ложное значение. Операция `||` — это булевская операция "ИЛИ": она вычисляется в истинное значение, если один из ее операндов истинный (или оба), и в ложное значение, когда оба операнда ложные. Наконец, унарная операция `!` выполняет булевскую операцию "НЕ": она дает `true`, если ее операнд ложный, и `false`, если операнд истинный. Вот примеры:

```
if ((x === 0 && y === 0) || !(z === 0)) {
    // x и y равны нулю или z не равно нулю
}
```

Полные сведения об упомянутых выше операциях приведены в разделе 4.10.

3.5. null и undefined

`null` — это ключевое слово языка, оцениваемое в особое значение, которое обычно применяется для указания на отсутствие значения. Использование операции `typeof` на `null` возвращает строку `"object"`, указывая на то, что `null` можно считать особым объектным значением, которое служит признаком "отсутствия объекта". Однако на практике `null` обычно рассматривается как единственный член собственного типа и может применяться с целью индикации "отсутствия значения" для чисел и строк, а также объектов. В большинстве языков программирования имеется эквивалент `null` из JavaScript: вы можете быть знакомы с `NULL`, `nil` или `None`.

В JavaScript есть и второе значение, которое указывает на отсутствие значения. Значение `undefined` представляет более глубокий вид отсутствия. Это значение переменных, которые не были инициализированы, и то, что получается при запрашивании значений свойств объекта или элементов массива, которые не существуют. Значение `undefined` также является возвращаемым значением функций, явно не возвращающих значение, и значением параметров функций, для которых аргументы не передавались. `undefined` представляет собой заранее определенную глобальную константу (не ключевое слово языка вроде `null`, хотя практически такое различие не особенно важно), которая инициализируется значением `undefined`. Если вы примените к значению `undefined` операцию `typeof`, то она возвратит `"undefined"`, указывая на то, что оно — единственный член специального типа.

Вопреки описанным отличиям `null` и `undefined` служат признаком отсутствия значения и часто могут использоваться взаимозаменяемо. Операция равенства `==` считает их равными. (Для их различения применяйте операцию строгого равенства `===`.) Оба являются ложными значениями: они ведут себя подобно `false`, когда требуется булевское значение. Ни `null`, ни `undefined` не имеют свойств либо методов. На самом деле использование `.` или `[]` для доступа к какому-то свойству или методу значений `null` и `undefined` приводят к ошибке типа `TypeError`.

Я полагаю, что undefined предназначено для представления системного, непредвиденного или похожего на ошибку отсутствия значения, а null — для представления программного, нормального или ожидаемого отсутствия значения. Когда могу, я избегаю применения null и undefined, но если мне нужно присвоить одно из этих значений переменной или свойству либо передать одно из них в функцию или возвратить из нее, то обычно использую null. Некоторые программисты стараются вообще избегать null и применять на его месте undefined везде, где только возможно.

3.6. Тип Symbol

Символы были введены в ES6, чтобы служить нестроковыми именами свойств. Для понимания символов вы должны знать, что фундаментальный тип Object в JavaScript определен как неупорядоченная коллекция свойств, где каждое свойство имеет имя и значение. Именами свойств обычно (а до ES6 единственно) являются строки. Но в ES6 и последующих версиях для такой цели можно использовать символы:

```
let strname = "string name";           // Строка для применения в
                                       // качестве имени свойства
let symname = Symbol("propname");      // Символ для использования
                                       // в качестве имени свойства
typeof strname                         // => "string": strname - строка
typeof symname                         // => "symbol": symname - символ
let o = {};                            // Создать новый объект
o[strname] = 1;         // Определить свойство со строковым именем
o[symname] = 2;         // Определить свойство с именем Symbol
o[strname]              // => 1: доступ к свойству со строковым именем
o[symname]              // => 2: доступ к свойству с символьным именем
```

Символы не располагают литеральным синтаксисом. Чтобы получить значение символа, понадобится вызвать функцию Symbol(), которая никогда не возвращает то же самое значение дважды, даже когда вызывается с таким же аргументом. Это означает, что если вы вызываете Symbol() для получения значения символа, то при добавлении к объекту нового свойства можете безопасно применять полученное значение символа как имя свойства, не беспокоясь о возможности перезаписывания существующего свойства с тем же именем. Аналогично, если вы используете символьные имена свойств и не разделяете их символы, тогда можете быть уверены в том, что другие модули кода в программе не будут непредумышленно перезаписывать ваши свойства.

На практике символы выступают в качестве механизма расширения языка. Когда в ES6 был представлен цикл for/of (см. подраздел 5.4.4) и итерируемые объекты (см. главу 12), возникла необходимость определить стандартный метод, который классы могли бы реализовывать, чтобы делать себя итерируемыми. Но стандартизация любого индивидуального строкового имени для этого итераторного метода привела бы к нарушению работы существующего кода, так что взамен было решено применять символьное имя. Как будет показано в главе 12,

Symbol.iterator представляет собой символьное значение, которое можно использовать в качестве имени метода, делая объект итерируемым.

Функция Symbol() принимает необязательный строковый аргумент и возвращает уникальное значение типа Symbol. Если вы предоставите строковый аргумент, то переданная в нем строка будет включена в выход метода toString() типа Symbol. Тем не менее, имейте в виду, что вызов Symbol() два раза с той же самой строкой даст два совершенно разных значения Symbol:

```
let s = Symbol("sym_x");
s.toString()                    // => "Symbol(sym_x)"
```

toString() — единственный интересный метод в экземплярах Symbol. Однако вы должны знать еще две функции, относящиеся к Symbol. Во время применения символов иногда вам желательно делать их закрытыми в своем коде, гарантируя то, что ваши свойства никогда не будут конфликтовать со свойствами, которые используются другим кодом. Тем не менее, в других случаях вам может требоваться определить значение Symbol и широко делиться им с прочим кодом. Например, это может быть ситуация, когда вы определяете какое-то расширение и хотите, чтобы в нем мог принимать участие другой код, как было с описанным ранее механизмом Symbol.iterator.

Для последнего сценария в JavaScript определен глобальный реестр символов. Функция Symbol.for() принимает строковый аргумент и возвращает значение Symbol, ассоциированное с предоставленной вами строкой. Если никакого значения Symbol со строкой пока не ассоциировано, тогда создается и возвращается новое значение Symbol, иначе возвращается существующее значение. То есть функция Symbol.for() совершенно отличается от функции Symbol(): Symbol() никогда не возвращает одно и то же значение дважды, но Symbol.for() всегда возвращает то же самое значение, когда вызывается с одной и той же строкой. Переданная функции Symbol.for() строка присутствует в выводе toString() для возвращенного значения Symbol; вдобавок ее можно извлечь, вызвав Symbol.keyFor() на возвращенном значении Symbol.

```
let s = Symbol.for("shared");
let t = Symbol.for("shared");
s === t                         // => true
s.toString()                    // => "Symbol(shared)"
Symbol.keyFor(t)                // => "shared"
```

3.7. Глобальный объект

В предшествующих разделах объяснялись элементарные типы и значения JavaScript. Объектные типы — объекты, массивы и функции — раскрываются в отдельных главах позже в книге. Но есть одно очень важное объектное значение, которое мы должны описать прямо сейчас. *Глобальный объект* — это обыкновенный объект JavaScript, который служит крайне важной цели: его свойствами являются глобально определенные идентификаторы, доступные программе JavaScript. Когда интерпретатор JavaScript запускается (или всякий раз, когда

веб-браузер загружает новую страницу), он создает новый глобальный объект и предоставляет ему начальный набор свойств, которые определяют:

- глобальные константы вроде `undefined`, `Infinity` и `NaN`;
- глобальные функции наподобие `isNaN()`, `parseInt()` (см. подраздел 3.9.2) и `eval()` (см. раздел 4.12);
- функции конструкторов вроде `Date()`, `RegExp()`, `String()`, `Object()` и `Array()` (см. подраздел 3.9.2);
- глобальные объекты наподобие `Math` и `JSON` (см. раздел 6.8).

Начальные свойства глобального объекта не являются зарезервированными словами, но заслуживают такого обращения. Некоторые глобальные свойства уже были описаны в настоящей главе. Большинство других будут раскрыты в других местах книги.

В Node глобальный объект имеет свойство с именем `global`, значением которого будет сам глобальный объект, поэтому в программах Node вы всегда можете ссылаться на глобальный объект по имени `global`.

В веб-браузерах глобальным объектом для всего кода JavaScript, содержащегося в окне браузера, служит объект `Window`, который представляет это окно. Глобальный объект `Window` располагает ссылающимся на самого себя свойством `window`, которое можно применять для ссылки на глобальный объект. Объект `Window` определяет основные глобальные свойства, а также довольно много других глобальных свойств, специфичных для веб-браузеров и кода JavaScript на стороне клиента. Потоки веб-воркеров (см. раздел 15.13) имеют другой глобальный объект, нежели объект `Window`, с которым они ассоциированы. Код в воркере может ссылаться на свой глобальный объект посредством `self`.

Наконец, в качестве стандартного способа ссылки на глобальный объект в любом контексте версия ES2020 определяет `globalThis`. К началу 2020 года данная возможность была реализована всеми современными браузерами и Node.

3.8. Неизменяемые элементарные значения и изменяемые объектные ссылки

В JavaScript существует фундаментальное отличие между элементарными значениями (`undefined`, `null`, булевские, числа и строки) и объектами (включая массивы и функции). Элементарные значения неизменяемы: никакого способа модифицировать элементарное значение не предусмотрено. Для чисел и булевских значений такое положение дел вполне очевидно — изменять значение числа даже не имеет смысла. Однако для строк это не настолько очевидно. Поскольку строки похожи на массивы символов, вы могли бы рассчитывать на наличие возможности изменять символ по указанному индексу. В действительности JavaScript не позволяет поступать так, и все строковые методы, которые выглядят как возвращающие модифицированную строку, на самом деле возвращают новое строковое значение.

Например:

```
let s = "hello";   // Начать с некоторого текста в нижнем регистре
s.toUpperCase();   // Возвращает "HELLO", но не изменяет s
s                  // => "hello": исходная строка не изменилась
```

Элементарные значения также сравниваются *по величине*: два значения будут одинаковыми, только если они имеют одну и ту же величину. Это выглядит справедливым для чисел, булевских значений, null и undefined: другого способа их сравнения попросту не существует. Тем не менее, для строк снова все не так очевидно. При сравнении двух отдельных строковых значений JavaScript трактует их как равные тогда и только тогда, когда они имеют одну и ту же длину и одинаковые символы по каждому индексу.

Объекты отличаются от элементарных значений. Прежде всего, они *изменяемы* — их значения можно модифицировать:

```
let o = { x: 1 };   // Начать с какого-нибудь объекта
o.x = 2;            // Модифицировать его, изменив значение свойства
o.y = 3;            // Еще раз модифицировать его, добавив новое свойство

let a = [1,2,3];    // Массивы также изменяемы
a[0] = 0;           // Изменить значение элемента массива
a[3] = 4;           // добавить новый элемент массива
```

Объекты не сравниваются по величине: два отдельных объекта не равны, даже когда имеют те же самые свойства и значения. И два отдельных массива не равны, даже если они содержат те же самые элементы в том же порядке:

```
let o = {x: 1}, p = {x: 1};  // Два объекта с одинаковыми свойствами
o === p                      // => false: отдельные объекты никогда не равны
let a = [], b = [];          // Два отдельных пустых массива
a === b                      // => false: отдельные массивы никогда не равны
```

Иногда объекты называют *ссылочными типами*, чтобы отличать их от элементарных типов JavaScript. В рамках такой терминологии объектные значения являются *ссылками*, и мы говорим, что объекты сравниваются *по ссылке*: два объектных значения будут одинаковыми, если и только если они *ссылаются* на тот же самый внутренний объект.

```
let a = [];     // Переменная a ссылается на пустой массив
let b = a;      // Теперь на тот же самый массив ссылается переменная b
b[0] = 1;       // Изменить массив, на который ссылается переменная b
a[0]            // => 1: изменение также будет видимым через переменную a
a === b         // => true: a и b ссылаются на тот же самый объект,
                //    поэтому они равны
```

В коде видно, что присваивание объекта (или массива) переменной просто присваивает ссылку: копия объекта не создается. Если вы хотите создать новую копию объекта или массива, тогда должны явно скопировать свойства объекта или элементы массива. В следующем примере демонстрируется использование цикла for (см. подраздел 5.4.3):

```
let a = ["a","b","c"]; // Массив, который мы хотим скопировать
let b = [];            // Отдельный массив, куда мы будем копировать
for(let i = 0; i < a.length; i++) {  // Для каждого из всех индексов
   b[i] = a[i];        // Копировать элемент a в b
}
let c = Array.from(b); //В ES6 копируйте массивы с помощью Array.from()
```

Подобным образом при желании сравнить два отдельных объекта или массива мы обязаны сравнивать их свойства или элементы. В приведенном ниже коде определяется функция для сравнения двух массивов:

```
function equalArrays(a, b) {
   if (a === b) return true;           // Идентичные массивы равны
   if (a.length !== b.length) return false; //Массивы разного размера
                                            // не равны
   for(let i = 0; i < a.length; i++) {//Цикл по всем элементам
      if (a[i] !== b[i]) return false; //Если любые элементы отличаются,
                                       //то массивы не равны
   }
   return true;                        // Иначе они равны
}
```

3.9. Преобразования типов

Язык JavaScript очень гибок в отношении типов значений, которые ему требуются. Мы видели это для булевских значений: когда интерпретатор JavaScript ожидает булевское значение, вы можете предоставить значение любого типа, и интерпретатор JavaScript при необходимости преобразует его. Одни значения ("истинные" значения) преобразуются в `true`, а другие ("ложные" значения) — в `false`. То же самое справедливо для остальных типов: если интерпретатор JavaScript пожелает иметь строку, тогда любое переданное вами значение будет преобразовано в строку. Если интерпретатор JavaScript желает число, то попробует преобразовать предоставленное вами значение в число (или в `NaN`, когда осмысленное преобразование невозможно).

Вот несколько примеров:

```
10 + " objects"     // => "10 objects": число 10 преобразуется в строку
"7" * "4"           // => 28: обе строки преобразуются в числа
let n = 1 - "x";    // n == NaN; строка "x" не может быть
                    //          преобразована в число
n + " objects"      //=>"NaN objects": NaN преобразуется в строку "NaN"
```

В табл. 3.2 приведена сводка, касающаяся того, как значения одного типа преобразуются в другой тип в JavaScript. Ячейки, помеченные полужирным, содержат преобразования, которые могут вас удивить. Пустые ячейки указывают на то, что в преобразованиях нет необходимости, поэтому они не делались.

Таблица 3.2. Преобразования типов JavaScript

Значение	Преобразование в строку	Преобразование в число	Преобразование в булевское значение
`undefined`	`"undefined"`	`NaN`	`false`
`null`	`"null"`	`0`	`false`
`true`	`"true"`	`1`	
`false`	`"false"`	`0`	
`""` (пустая строка)		`0`	**`false`**
`"1.2"` (непустая, числовое содержимое)		`1.2`	`true`
`"one"` (непустая, нечисловое содержимое)		`NaN`	`true`
`0`	`"0"`		**`false`**
`-0`	`"0"`		**`false`**
`1` (конечное, ненулевое)	`"1"`		`true`
`Infinity`	`"Infinity"`		`true`
`-Infinity`	`"-Infinity"`		`true`
`NaN`	`"NaN"`		**`false`**
`{}` (любой объект)	См. подраздел 3.9.3	См. подраздел 3.9.3	`true`
`[]` (пустой массив)	`""`	`0`	`true`
`[9]` (один числовой элемент)	`"9"`	`9`	`true`
`['a']` (любой другой массив)	Используйте метод `join()`	`NaN`	`true`
`function(){}` (любая функция)	См. подраздел 3.9.3	`NaN`	`true`

Показанные в табл. 3.2 преобразования между элементарными типами относительно прямолинейны. Преобразование в булевские значения уже обсуждалось в разделе 3.4. Преобразование в строки четко определены для всех элементарных значений. Преобразование в числа лишь немного сложнее. Строки, которые могут быть разобраны как числа, преобразуются в эти числа. Начальные и конечные пробелы разрешены, но наличие любых начальных или конечных непробельных символов, не являющихся частью числового литерала, приведет к тому, что результатом преобразования строки в число окажется `NaN`. Некоторые преобразования в числа могут показаться удивительными: `true` преобразуется в 1, а `false` и пустая строка — в 0.

Преобразование объектов в элементарные типы кое в чем сложнее и будет рассматриваться в подразделе 3.9.3.

3.9.1. Преобразования и равенство

В JavaScript есть две операции, которые проверяют, равны ли два значения. "Операция строгого равенства", ===, не считает свои операнды равными, если они не относятся к одному и тому же типу, и при написании кода она почти всегда будет правильным выбором. Но поскольку язык JavaScript настолько гибок в плане преобразований типов, в нем определена также и операция == с гибкой формулировкой равенства. Скажем, все нижеследующие сравнения дают `true`:

```
null == undefined  // => true: эти два значения трактуются как равные.
"0" == 0   // => true: перед сравнением строка преобразуется в число.
0 == false         // => true: перед сравнением булевское значение
                   //          преобразуется в число.
"0" == false       // => true: перед сравнением оба операнда
                   //          преобразуются в 0!
```

В подразделе 4.9.1 объясняется, какие именно преобразования выполняет операция ==, чтобы выяснить, должны ли два значения считаться равными.

Имейте в виду, что возможность преобразования одного значения в другое вовсе не подразумевает равенство этих двух значений. Например, если `undefined` используется там, где ожидается булевское значение, то оно будет преобразовано в `false`. Но это не означает, что `undefined == false`. Операции и операторы JavaScript ожидают значения разнообразных типов и выполняют преобразования в них. Оператор `if` преобразует `undefined` в `false`, но операция == никогда не пытается преобразовывать свои операнды в булевские значения.

3.9.2. Явные преобразования

Хотя JavaScript выполняет многие преобразования типов автоматически, иногда вам может понадобиться явное преобразование или, возможно, вы предпочитаете делать преобразования явно, поддерживая ясность своего кода.

Простейший способ выполнения явных преобразований типов предусматривает применение функций `Boolean()`, `Number()` и `String()`:

```
Number("3")        // => 3
String(false)      // => "false": либо используйте false.toString()
Boolean([])        // => true
```

Любое значение кроме `null` и `undefined` имеет метод `toString()`, результат которого обычно будет таким же, как возвращаемый функцией `String()`.

В качестве ремарки отметим, что функции `Boolean()`, `Number()` и `String()` можно также вызывать посредством `new` как конструкторы. В случае их использования подобным образом вы получите "объект-оболочку", который ведет себя точно как элементарное булевское, числовое или строковое значение. Такие объекты-оболочки являются историческим пережитком с самых первых лет становления JavaScript, и по правде говоря, никогда не существовало веских причин применять их.

Определенные операции JavaScript выполняют неявное преобразование типов и временами используются явно в целях преобразования типов. Если одним операндом операции + является строка, тогда другой операнд преобразуется в строку. Унарная операция + преобразует свой операнд в число. Унарная операция ! преобразует свой операнд в булевское значение и инвертирует его. Перечисленные факты приводят к следующим идиомам преобразования типов, которые вы могли встречать в коде:

```
x + ""      // => String(x)
+x          // => Number(x)
x-0         // => Number(x)
!!x         // => Boolean(x): обратите внимание на два символа !
```

Форматирование и разбор чисел — распространенные задачи в компьютерных программах, и в JavaScript имеются специализированные функции и методы, которые предлагают более точный контроль над преобразованиями чисел в строки и строк в числа.

Метод `toString()`, определенный в классе `Number`, принимает необязательный аргумент, где задается основание системы счисления для преобразования. Если вы не укажете этот аргумент, то преобразование будет делаться по основанию 10. Однако вы можете также преобразовывать числа в другие системы счисления (с основаниями между 2 и 36), например:

```
let n = 17;
let binary = "0b" + n.toString(2);   // binary == "0b10001"
let octal = "0o" + n.toString(8);    // octal == "0o21"
let hex = "0x" + n.toString(16);     // hex == "0x11"
```

При работе с финансовыми или научными данными вас может интересовать преобразование чисел способами, которые предоставляют контроль над количеством десятичных разрядов или количеством значащих цифр в выводе, либо же вам нужно управлять тем, применяется ли экспоненциальная запись. В классе `Number` определены три метода для преобразований чисел в строки такого рода. Метод `toFixed()` преобразует число в строку с указанным количеством цифр после десятичной точки. Он никогда не использует экспоненциальную запись. Метод `toExponential()` преобразует число в строку, применяя экспоненциальную запись с одной цифрой перед десятичной точкой и указанным количеством цифр после десятичной точки (это значит, что количество значащих цифр на единицу больше заданной вами величины). Метод `toPrecision()` преобразует число в строку с указанным вами количеством значащих цифр. Он использует экспоненциальную запись, если количество цифр недостаточно велико, чтобы полностью отобразить целую часть числа. Обратите внимание, что все три метода в зависимости от ситуации округляют хвостовые цифры или дополняют нулями. Взгляните на следующие примеры:

```
let n = 123456.789;
n.toFixed(0)          // => "123457"
n.toFixed(2)          // => "123456.79"
n.toFixed(5)          // => "123456.78900"
n.toExponential(1)    // => "1.2e+5"
```

```
n.toExponential(3)          // => "1.235e+5"
n.toPrecision(4)            // => "1.235e+5"
n.toPrecision(7)            // => "123456.8"
n.toPrecision(10)           // => "123456.7890"
```

В дополнение к показанным здесь методам форматирования чисел класс `Intl.NumberFormat` определяет более универсальный, интернационализированный метод форматирования чисел. За деталями обращайтесь в подраздел 11.7.1.

Если вы передадите строку функции преобразования `Number()`, то она попытается разобрать ее как целочисленный литерал или литерал с плавающей точкой. Данная функция работает только с десятичными целыми числами и не допускает наличия конечных символов, которые не относятся к литералу. Функции `parseInt()` и `parseFloat()` (это глобальные функции, не методы какого-то класса) более гибкие. `parseInt()` разбирает только целые числа, в то время как `parseFloat()` — целые числа и числа с плавающей точкой. Если строка начинается с `0x` или `0X`, то `parseInt()` интерпретирует ее как шестнадцатеричное число. Оба метода, `parseInt()` и `parseFloat()`, пропускают начальные пробельные символы, разбирают столько числовых символов, сколько могут, и игнорируют все, что находится за ними. Если первый непробельный символ не является частью допустимого числового литерала, тогда они возвращают `NaN`:

```
parseInt("3 blind mice")    // => 3
parseFloat(" 3.14 meters")  // => 3.14
parseInt("-12.34")          // => -12
parseInt("0xFF")            // => 255
parseInt("0xff")            // => 255
parseInt("-0XFF")           // => -255
parseFloat(".1")            // => 0.1
parseInt("0.1")             // => 0
parseInt(".1")              // => NaN: целые числа не могут начинаться
                            //         с символа точки
parseFloat("$72.47")        // => NaN: числа не могут начинаться
                            //         с символа доллара
```

Функция `parseInt()` принимает необязательный второй аргумент, указывающий основание системы счисления числа, которое подлежит разбору. Допустимые значения находятся между 2 и 36. Вот примеры:

```
parseInt("11", 2)           // => 3: (1*2 + 1)
parseInt("ff", 16)          // => 255: (15*16 + 15)
parseInt("zz", 36)          // => 1295: (35*36 + 35)
parseInt("077", 8)          // => 63: (7*8 + 7)
parseInt("077", 10)         // => 77: (7*10 + 7)
```

3.9.3. Преобразования объектов в элементарные значения

В предшествующих разделах было показано, как явно преобразовывать значения одного типа в другой, и объяснялись неявные преобразования значений из одного элементарного типа в другой элементарный тип, выполняемые

JavaScript. В этом разделе раскрываются сложные правила, которые JavaScript применяет для преобразования объектов в элементарные значения. Обсуждение может показаться длинным и неясным, так что если вы впервые читаете главу, тогда смело можете переходить сразу к разделу 3.10.

Одна из причин сложности преобразований объектов в элементарные значения связана с тем, что некоторые типы объектов имеют несколько элементарных представлений. Скажем, объекты дат могут быть представлены как строки или как числовые отметки времени. В спецификации JavaScript определены три фундаментальных алгоритма для преобразования объектов в элементарные значения.

prefer-string

Этот алгоритм возвращает элементарное значение, отдавая предпочтение строке, если такое преобразование возможно.

prefer-number

Этот алгоритм возвращает элементарное значение, отдавая предпочтение числу, если такое преобразование возможно.

no-preference

Этот алгоритм не отдает никаких предпочтений относительно того, какой тип элементарного значения желателен, и классы могут определять собственные преобразования. Все встроенные типы JavaScript, исключая `Date`, реализуют данный алгоритм как *prefer-number*. Класс `Date` реализует его как *prefer-string*.

Реализация перечисленных алгоритмов преобразования объектов в элементарные значения объясняется в конце раздела. Но сначала нужно выяснить, как такие алгоритмы используются в JavaScript.

Преобразования объектов в булевские значения

Преобразования объектов в булевские значения тривиальны: все объекты преобразуются в `true`. Обратите внимание, что такое преобразование не требует использования описанных выше алгоритмов для преобразования объектов в элементарные значения, и оно буквально применяется ко *всем* объектам, включая пустые массивы и даже объекты-оболочки `new Boolean(false)`.

Преобразования объектов в строковые значения

Когда объект нужно преобразовать в строку, интерпретатор JavaScript сначала преобразует его в элементарное значение, используя алгоритм *prefer-string*, а затем при необходимости преобразует результирующее элементарное значение в строку, следуя правилам из табл. 3.2.

Преобразование такого вида происходит, например, в случае передачи объекта встроенной функции, которая ожидает строковый аргумент, при вызове `String()` в качестве функции преобразования и в ситуации, когда объекты интерполируются внутрь шаблонных литералов (см. подраздел 3.3.4).

Преобразования объектов в числовые значения

Когда объект нужно преобразовать в число, интерпретатор JavaScript сначала преобразует его в элементарное значение, используя алгоритм *prefer-number*, а затем при необходимости преобразует результирующее элементарное значение в число, следуя правилам из табл. 3.2.

Подобным образом преобразуют объекты в числа встроенные функции и методы JavaScript, ожидающие числовые аргументы, а также большинство операций JavaScript (с описанными ниже исключениями), которые ожидают числовые операнды.

Преобразования в операциях особого случая

Операции подробно рассматриваются в главе 4. Здесь мы объясним операции особого случая, которые не используют описанные ранее базовые преобразования объектов в строки и числа.

Операция + в JavaScript выполняет сложение чисел и конкатенацию строк. Если любой из ее операндов оказывается объектом, то интерпретатор JavaScript преобразует его в элементарное значение с применением алгоритма *no-preference*. После получения двух элементарных значений проверяются их типы. Если один из аргументов является строкой, тогда другой преобразуется в строку и выполняется конкатенация строк. Иначе оба аргумента преобразуются в числа и складываются.

Операции == и != проверяют на предмет равенства и неравенства свободным способом, который допускает преобразование типов. Если один операнд представляет собой объект, а другой — элементарное значение, тогда эти операции преобразуют объект в элементарное значение, используя алгоритм *no-preference*, и затем сравнивают два элементарных значения.

Наконец, операции отношения <, <=, > и >= сравнивают порядок своих операндов и могут применяться для сравнения как чисел, так и строк. Если любой из операндов является объектом, то он преобразуется в элементарное значение с использованием алгоритма *prefer-number*. Тем не менее, обратите внимание, что в отличие от преобразования объектов в числа элементарные значения, возвращенные преобразованием *prefer-number*, впоследствии не преобразуются в числа.

Отметим, что числовое представление объектов `Date` поддерживает осмысленное сравнение с помощью < и >, но строковое представление — нет. Алгоритм *no-preference* преобразует объекты `Date` в строки, поэтому тот факт, что интерпретатор JavaScript применяет для указанных операций алгоритм *prefer-number*, означает возможность их использования для сравнения порядка двух объектов `Date`.

Методы `toString()` и `valueOf()`

Все объекты наследуют два метода преобразования, которые применяются преобразованиями объектов в элементарные значения, и прежде чем можно будет объяснять алгоритмы преобразования *prefer-string*, *prefer-number* и *no-preference*, нужно взглянуть на эти два метода.

Первым методом является `toString()`, задача которого — возвратить строковое представление объекта. Стандартный метод `toString()` возвращает не особо интересное значение (хотя мы сочтем его полезным в подразделе 14.4.3):

```
({x: 1, y: 2}).toString()       // => "[object Object]"
```

Многие классы определяют более специфичные версии метода `toString()`. Скажем, метод `toString()` класса `Array` преобразует каждый элемент массива в строку и объединяет результирующие строки вместе, разделяя их запятыми. Метод `toString()` класса `Function` преобразует определяемые пользователем функции в строки исходного кода JavaScript. Класс `Date` определяет метод `toString()`, который возвращает пригодную для чтения человеком (и поддающуюся разбору интерпретатором JavaScript) строку с датой и временем. Класс `RegExp` определяет метод `toString()`, преобразующий объект `RegExp` в строку, которая выглядит подобно литералу `RegExp`:

```
[1,2,3].toString()              // => "1,2,3"
(function(x) { f(x); }).toString()  // => "function(x) { f(x); }"
/\d+/g.toString()               // => "/\\d+/g"
let d = new Date(2020,0,1);
d.toString()                    //=>"Wed Jan 01 2020 00:00:00 GMT-0800
                                //   (Pacific Standard Time)"
```

Другая функция преобразования объектов называется `valueOf()`. Задача этого метода определена менее четко: предполагается, что он должен преобразовывать объект в элементарное значение, которое представляет объект, если такое элементарное значение существует. Объекты являются составными значениями, причем большинство объектов на самом деле не могут быть представлены единственным элементарным значением, а потому стандартный метод `valueOf()` возвращает сам объект вместо элементарного значения. Классы-оболочки, такие как `String`, `Number` и `Boolean`, определяют методы `valueOf()`, которые просто возвращают содержащееся внутри элементарное значение. Массивы, функции и регулярные выражения наследуют стандартный метод. Вызов `valueOf()` для экземпляров упомянутых типов возвращает сам объект. Класс `Date` определяет метод `valueOf()`, возвращающий дату в ее внутреннем представлении — количество миллисекунд, прошедших с момента 1 января 1970 года:

```
let d = new Date(2010, 0, 1); //1 января 2010 года (тихоокеанское время)
d.valueOf()                   // => 1262332800000
```

Алгоритмы преобразования объектов в элементарные значения

После обсуждения методов `toString()` и `valueOf()` теперь мы можем схематично объяснить, как работают три алгоритма преобразования объектов в элементарные значения (полные детали будут приведены в подразделе 14.4.7).

- Алгоритм prefer-string сначала испытывает метод `toString()`. Если он определен и возвращает элементарное значение, тогда интерпретатор JavaScript использует это элементарное значение (даже когда оно отлично от строки!). Если `toString()` не существует или возвращает объект, то

интерпретатор JavaScript испытывает метод `valueOf()`. Если он определен и возвращает элементарное значение, тогда интерпретатор JavaScript использует это значение. Иначе преобразование терпит неудачу с выдачей ошибки типа (`TypeError`).

- Алгоритм prefer-number работает аналогично алгоритму prefer-string, но первым испытывается `valueOf()`, а вторым — `toString()`.
- Алгоритм no-preference зависит от класса преобразуемого объекта. Если объект относится к классу `Date`, тогда интерпретатор JavaScript применяет алгоритм prefer-string. Для любого другого объекта интерпретатор JavaScript использует алгоритм prefer-number.

Описанные здесь правила справедливы для всех встроенных типов JavaScript и по умолчанию принимаются для любых классов, которые вы определяете самостоятельно. В подразделе 14.4.7 будет объяснено, как определять собственные алгоритмы преобразования объектов в элементарные значения для создаваемых вами классов.

В завершение данной темы полезно отметить, что детали преобразования *prefer-number* объясняют, почему пустые массивы преобразуются в число 0 и одноэлементные массивы также могут быть преобразованы в числа:

```
Number([])      // => 0: неожиданно!
Number([99])    // => 99: правда?
```

Преобразование объектов в числа сначала преобразует объект в элементарное значение с применением алгоритма *prefer-number*, после чего преобразует результирующее элементарное значение в число. Алгоритм *prefer-number* первым испытывает `valueOf()` и затем возвращается к `toString()`. Но класс `Array` наследует стандартный метод `valueOf()`, который не возвращает элементарное значение. Таким образом, когда мы пытаемся преобразовать массив в число, то в итоге получаем вызов метода `toString()` массива. Пустой массив преобразуется в пустую строку, а пустая строка преобразуется в число 0. Массив с единственным элементом преобразуется в ту же строку, что и этот один элемент. Если массив содержит единственное число, то оно преобразуется в строку и затем обратно в число.

3.10. Объявление и присваивание переменных

Одной из наиболее фундаментальных методик в программировании является использование имен — или *идентификаторов* — для представления значений. Привязка имени к значению дает нам способ ссылки на это значение и его применение в программах, которые мы пишем. Поступая так, мы обычно говорим, что присваиваем значение *переменной*. Термин "переменная" подразумевает возможность присваивания новых значений: что значение, ассоциированное с переменной, может варьироваться в ходе выполнения программы. Если мы присваиванием значение имени на постоянной основе, тогда такое имя будет называться *константой*, а не переменной.

Прежде чем переменную или константу можно будет использовать в программе JavaScript, ее необходимо *объявить*. В ES6 и последующих версиях объявление делается с помощью ключевых слов `let` и `const`, как объясняется ниже. До выхода ES6 переменные объявлялись посредством `var`, что в большей степени отличается и рассматривается позже в разделе.

3.10.1. Объявление с помощью `let` и `const`

В современном JavaScript (ES6 и последующие версии) переменные объявляются с помощью ключевого слова `let`, например:

```
let i;
let sum;
```

В одном операторе `let` можно также объявлять сразу несколько переменных:

```
let i, sum;
```

Хорошая практика программирования предусматривает присваивание начальных значений переменных при их объявлении, когда это возможно:

```
let message = "hello";
let i = 0, j = 0, k = 0;
let x = 2, y = x*x;      // В инициализаторах можно использовать
                         // ранее объявленные переменные
```

Если вы не укажете начальное значение для переменной в операторе `let`, то переменная объявится, но ее значением будет `undefined`, пока где-то в коде не произойдет присваивание ей значения.

Чтобы объявить константу, а не переменную, применяйте `const` вместо `let`. Ключевое слово `const` похоже на `let`, но вы обязаны инициализировать константу при ее объявлении:

```
const H0 = 74;           // Постоянная Хаббла ((км/с)/Мпк)
const C = 299792.458;    // Скорость света в вакууме (км/с)
const AU = 1.496E8;      // Астрономическая единица: расстояние
                         // до Солнца (км)
```

Как следует из названия, значения констант изменяться не может и любая попытка изменения приводит к генерации ошибки `TypeError`.

Общепринятое (но не универсальное) соглашение предусматривает объявление констант с использованием имен, содержащих все прописные буквы, наподобие `H0` или `HTTP_NOT_FOUND`, как способ их различения от переменных.

Когда использовать `const`?

Существуют два подхода к применению ключевого слова `const`. Первый подход заключается в том, чтобы использовать `const` только для фундаментально неизменяемых значений, таких как показанные выше физические постоянные, номера версий программ или байтовые последовательности, идентифицирующие файловые типы. Второй подход формально устанавливает, что многие так называемые переменные в программе на самом деле никогда не изменяются в ходе ее выполнения.

При таком подходе мы объявляем все с помощью const и затем, если обнаруживаем, что в действительности хотим позволить значению меняться, то переключаем объявление на let. Этот подход помогает избежать ошибок, исключая случайные изменения переменных, которые не входили в наши намерения.

При одном подходе мы применяем const только для значений, которые *не должны* изменяться. При другом подходе мы используем const для любого значения, которое не будет изменяться. В своем коде я отдаю предпочтение первому подходу.

В главе 5 речь пойдет об операторах циклов for, for/in и for/of в JavaScript. Каждый из указанных циклов включает переменную цикла, которая получает новое значение, присваиваемое ей на каждой итерации цикла. JavaScript позволяет нам объявлять переменную цикла как часть синтаксиса самого цикла, и это еще один распространенный способ применения let:

```
for(let i = 0, len = data.length; i < len; i++) console.log(data[i]);
for(let datum of data) console.log(datum);
for(let property in object) console.log(property);
```

Может показаться удивительным, но при объявлении "переменных" циклов for/in и for/of допускается использовать также и const, если в теле цикла им не присваивается новое значение. В таком случае объявление const просто говорит о том, что значение будет константой в течение одной итерации цикла:

```
for(const datum of data) console.log(datum);
for(const property in object) console.log(property);
```

Область видимости переменных и констант

Область видимости переменной представляет собой часть исходного кода программы, в которой переменная определена. Переменные и константы, объявленные с помощью let и const, имеют *блочную область видимости*. Другими словами, они определены только внутри блока кода, в котором находятся оператор let или const. Определения классов и функций JavaScript являются блоками, равно как и тела операторов if/else, циклов while, for и т.д. Грубо говоря, если переменная или константа объявляется внутри набора фигурных скобок, то эти фигурные скобки ограничивают область кода, где переменная или константа определена (хотя, конечно же, нельзя ссылаться на переменную или константу из строк кода, который выполняется до оператора let или const, объявляющего переменную или константу). Переменные и константы, объявленные как часть цикла for, for/in или for/of, имеют в качестве области видимости тело цикла, даже если формально они появляются за рамками фигурных скобок.

Когда объявление находится на верхнем уровне, за пределами любых блоков кода, мы говорим, что оно представляет *глобальную* переменную или константу и имеет глобальную область видимости. В Node и в модулях JavaScript стороны клиента (см. главу 10) областью видимости глобальной переменной являет-

ся файл, в котором она определена. Однако в традиционном JavaScript стороны клиента областью видимости глобальной переменной будет HTML-документ, где она определена. То есть, если в одном элементе <script> объявляется глобальная переменная или константа, то она считается определенной во всех элементах <script> в данном документе (или, во всяком случае, во всех сценариях, которые выполняются после выполнения оператора let или const).

Повторное объявление

Применение одного и того же имени в более чем одном объявлении let или const внутри той же области видимости является синтаксической ошибкой. Объявлять новую переменную с тем же именем во вложенной области видимости разрешено (но на практике лучше так не поступать):

```
const x = 1;          // Объявить x как глобальную константу
if (x === 1) {
   let x = 2;         // Внутри блока x может ссылаться на другое значение
   console.log(x); // Выводится 2
}
console.log(x);       // Выводится 1: мы снова вернулись
                      // в глобальную область видимости
let x = 3;            // ОШИБКА! Синтаксическая ошибка при попытке
                      // повторного объявления x
```

Объявление и типы

Если вы привыкли к статически типизированным языкам, таким как C или Java, то можете подумать, что основная цель объявления переменной заключается в указании типа значений, которые могут ей присваиваться. Но, как вы видели, с объявлениями переменных JavaScript никакие типы не ассоциированы[2]. Переменная JavaScript может хранить значение любого типа. Например, в JavaScript совершенно законно (но обычно считается плохим стилем программирования) присвоить переменной число, а позже строку:

```
let i = 10;
i = "ten";
```

3.10.2. Объявление переменных с помощью var

В версиях JavaScript, предшествующих ES6, единственный способ объявления переменной предусматривал использование ключевого слова var, а объявлять константы было вообще невозможно. Синтаксис var похож на синтаксис let:

```
var x;
var data = [], count = data.length;
for(var i = 0; i < count; i++) console.log(data[i]);
```

[2] Существуют расширения JavaScript вроде TypeScript и Flow (см. раздел 17.8), которые позволяют указывать типы как часть объявлений переменных посредством синтаксиса вида let x: number = 0;.

Хотя var и let имеют одинаковый синтаксис, существуют важные отличия в том, как они работают.

- Переменные, объявленные с помощью var, не имеют блочной области видимости. Взамен область видимости таких переменных распространяется на тело содержащей функции независимо от того, насколько глубоко их объявления вложены внутрь этой функции.
- Если вы применяете var вне тела функции, то объявляется глобальная переменная. Но глобальные переменные, объявленные посредством var, отличаются от глобальных переменных, объявленных с использованием let, одним важным аспектом. Глобальные переменные, объявленные с помощью var, реализуются в виде свойств глобального объекта (см. раздел 3.7). На глобальный объект можно ссылаться как на globalThis. Таким образом, если вы записываете var x = 2; за пределами функции, то это подобно записи globalThis.x = 2;. Тем не менее, следует отметить, что аналогия не идеальна: свойства, созданные из глобальных объявлений var, не могут быть удалены посредством операции delete (см. подраздел 4.13.4). Глобальные переменные и константы, объявленные с применением let и const, не являются свойствами глобального объекта.
- В отличие от переменных, объявленных с помощью let, с использованием var вполне законно объявлять ту же самую переменную много раз. И поскольку переменные var имеют область видимости функции, а не блока, фактически общепринято делать повторное объявление такого рода. Переменная i часто применяется для целочисленных значений особенно в качестве индексной переменной циклов for. В функции с множеством циклов for каждый цикл обычно начинается с for(var i = 0; ... Из-за того, что var не ограничивает область видимости этих переменных телом цикла, каждый цикл (безопасно) повторно объявляет и заново инициализирует ту же самую переменную.
- Одна из самых необычных особенностей объявлений var известна как подъем (hoisting). Когда переменная объявляется с помощью var, объявление поднимается к верхушке объемлющей функции. Инициализация переменной остается там, где вы ее записали, но определение переменной перемещается к верхушке функции. Следовательно, переменные, объявленные посредством var, можно использовать безо всяких ошибок где угодно в объемлющей функции. Если код инициализации пока еще не выполнен, тогда значением переменной может быть undefined, но вы не получите ошибку в случае работы с переменной до ее инициализации. (Это может стать источником ошибок и одной из нежелательных характеристик, которую исправляет let: если вы объявите переменную с применением let, но попытаетесь ее использовать до выполнения оператора let, то получите действительную ошибку, а не просто увидите значение undefined.)

Использование необъявленных переменных

В строгом режиме (см. подраздел 5.6.3) при попытке использовать необъявленную переменную во время запуска кода вы получите ошибку ссылки. Однако за рамками строгого режима, если вы присвоите значение имени, которое не было объявлено с применением `let`, `const` или `var`, то в итоге создадите новую глобальную переменную. Переменная будет глобальной независимо от того, насколько глубоко она вложена внутри функций и блоков кода, что почти наверняка не будет тем, что вы хотите, подвержено ошибкам и одной из наилучших причин использования строгого режима!

Глобальные переменные, которые созданы описанным дополнительным способом, похожи на глобальные переменные, объявленные посредством `var`: они определяют свойства глобального объекта. Но в отличие от свойств, определенных надлежащими объявлениями `var`, такие свойства *могут* быть удалены с помощью операции `delete` (см. подраздел 4.13.4).

3.10.3. Деструктурирующее присваивание

В ES6 реализован своего рода составной синтаксис объявления и присваивания, известный как *деструктурирующее присваивание* (destructuring assignment). При деструктурирующем присваивании значение с правой стороны знака равенства является массивом или объектом ("структурированное" значение), а с левой стороны указывается одна или большее количество имен переменных с применением синтаксиса, который имитирует литеральный синтаксис массивов или объектов. Когда происходит деструктурирующее присваивание, из значения справа извлекается ("деструктурируется") одно или большее количество значений и сохраняется в переменных, имена которых указаны слева. Деструктурирующее присваивание, вероятно, чаще всего используется для инициализации переменных как части оператора объявления `const`, `let` или `var`, но также может делаться в обыкновенных выражениях присваивания (с переменными, которые уже объявлены). Вдобавок, как мы увидим в подразделе 8.3.5, деструктуризация может применяться при определении параметров функции.

Ниже приведены простые деструктурирующие присваивания с применением массивов значений:

```
let [x,y] = [1,2];      // То же, что и let x=1, y=2
[x,y] = [x+1,y+1];      // То же, что и x = x + 1, y = y + 1
[x,y] = [y,x];          // Поменять местами значения двух переменных
[x,y]                   // => [3,2]: инкрементированные и поменявшиеся
                        //           местами значения
```

Обратите внимание, насколько деструктурирующее присваивание облегчает работу с функциями, которые возвращают массивы значений:

```
// Преобразует координаты [x,y] в полярные координаты [r,theta]
function toPolar(x, y) {
   return [Math.sqrt(x*x+y*y), Math.atan2(y,x)];
}
```

```
// Преобразует полярные координаты в декартовы
function toCartesian(r, theta) {
   return [r*Math.cos(theta), r*Math.sin(theta)];
}
let [r,theta] = toPolar(1.0, 1.0);  //r == Math.sqrt(2); theta == Math.PI/4
let [x,y] = toCartesian(r,theta);//[x, y] == [1.0, 1,0]
```

Мы видели, что переменные и константы могут объявляться как часть разнообразных циклов for. В данном контексте тоже можно использовать деструктуризацию переменных. Далее показан код, который проходит в цикле по парам "имя/значение" всех свойств объекта и применяет деструктурирующее присваивание для преобразования таких пар из двухэлементных массивов в отдельные переменные:

```
let o = { x: 1, y: 2 };                    // Объект, по которому будет
                                           // делаться проход в цикле
for(const [name, value] of Object.entries(o)) {
   console.log(name, value);               // Выводится "x 1" и "y 2"
}
```

Количество переменных слева от деструктурирующего присваивания не обязано совпадать с количеством элементов массива справа. Избыточные переменные слева устанавливаются в undefined, а избыточные значения справа игнорируются. Список переменных слева может включать дополнительные запятые, чтобы пропускать определенные значения справа:

```
let [x,y] = [1];                           // x == 1; y == undefined
[x,y] = [1,2,3];                           // x == 1; y == 2
[,x,,y] = [1,2,3,4];                       // x == 2; y == 4
```

Если при деструктуризации массива вы хотите собрать все неиспользуемые или оставшиеся значения в единственную переменную, тогда применяйте три точки (...) перед последним именем переменной с левой стороны:

```
let [x, ...y] = [1,2,3,4];                 // y == [2,3,4]
```

Три точки будут снова использоваться подобным образом в подразделе 8.3.2 для указания на то, что все оставшиеся аргументы функции должны собираться в одиночный массив.

Деструктурирующее присваивание можно применять с вложенными массивами. В таком случае левая сторона присваивания должна выглядеть похожей на вложенный литерал массива:

```
let [a, [b, c]] = [1, [2,2.5], 3];   // a == 1; b == 2; c == 2.5
```

Важная особенность деструктурирующего присваивания заключается в том, что на самом деле наличие массива вовсе не обязательно! С правой стороны присваивания можно использовать любой *итерируемый* объект (см. главу 12); любой объект, который может применяться с циклом for/of (см. подраздел 5.4.4), также можно деструктурировать:

```
let [first, ...rest] = "Hello";            // first == "H";
                                           // rest == ["e","l","l","o"]
```

Деструктурирующее присваивание может выполняться и в случае, когда с правой стороны указано объектное значение. Тогда левая сторона присваивания выглядит подобно объектному литералу, представляя собой разделенный запятыми список имен переменных в фигурных скобках:

```
let transparent = {r: 0.0, g: 0.0, b: 0.0, a: 1.0};  // Цвет RGBA
let {r, g, b} = transparent;       // r == 0.0; g == 0.0; b == 0.0
```

В следующем примере глобальные функции объекта Math копируются в переменные, что позволяет упростить код, выполняющий крупный объем тригонометрических вычислений:

```
// То же, что и const sin=Math.sin, cos=Math.cos, tan=Math.tan
const {sin, cos, tan} = Math;
```

Обратите внимание, что здесь объект Math имеет гораздо больше свойств, чем те три, которые были деструктурированы в отдельные переменные. Свойства, имена которых не указаны, попросту игнорируются. Если бы слева от этого присваивания присутствовала переменная с именем, не являющимся свойством Math, то она получила бы значение undefined.

В каждом из приведенных примеров деструктуризации объектов мы выбирали имена переменных, которые совпадали с именами свойств деструктурируемого объекта. Такой подход делает синтаксис простым и легким для понимания, но поступать так вовсе не обязательно. Каждый идентификатор слева от деструктурирующего присваивания объекта также может быть парой идентификаторов с запятой в качестве разделителя, где первый идентификатор представляет собой имя свойства, значение которого присваивается, а второй — имя присваиваемой переменной:

```
// То же, что и const cosine = Math.cos, tangent = Math.tan;
const { cos: cosine, tan: tangent } = Math;
```

Я считаю, что синтаксис деструктуризации объектов становится слишком сложным, когда имена переменных и свойств не совпадают, и стараюсь избегать сокращения, как в показанном выше коде. Если вы решите использовать его, тогда запомните, что имена свойств всегда находятся слева от двоеточия как в объектных литералах, так и с левой стороны деструктурирующего присваивания объектов.

Деструктурирующее присваивание еще более усложняется, когда применяется с вложенными объектами, массивами объектов или объектами массивов, но следующий код вполне законен:

```
let points = [{x: 1, y: 2}, {x: 3, y: 4}]; // Массив из двух объектов,
                                           // представляющих точку,
let [{x: x1, y: y1}, {x: x2, y: y2}] = points;  // деструктурируется
                                                // в 4 переменных.
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4)  // => true
```

Или вместо деструктуризации массива объектов мы могли бы деструктурировать объект массивов:

```
let points = { p1: [1,2], p2: [3,4] };  // Объект с 2 свойствами,
                                        // которые являются массивами,
let { p1: [x1, y1], p2: [x2, y2] } = points;  // деструктурируется
                                              // в 4 переменных.
(x1 === 1 && y1 === 2 && x2 === 3 && y2 === 4) // => true
```

Сложный синтаксис деструктуризации вроде этого может оказаться трудным для записи и восприятия, и возможно будет лучше записывать присваивания явно с помощью традиционного кода наподобие `let x1 = points.p1[0];`.

Понимание сложной деструктуризации

Если вам придется работать с кодом, в котором используются сложные деструктурирующие присваивания, то имеется полезная закономерность, способная помочь разобраться в запутанных случаях. Сначала подумайте об обыкновенном присваивании (с единственным значением). После того, как присваивание сделано, вы можете взять имя переменной с левой стороны присваивания и применять его в качестве выражения в своем коде, где оно будет оценено в любое значение, которое вы присвоили. То же самое справедливо в отношении деструктурирующего присваивания. Левая сторона деструктурирующего присваивания выглядит подобно литералу массива или объектному литералу (см. подраздел 6.2.1 и раздел 6.10). После выполнения присваивания левая сторона фактически будет служить допустимым литералом массива или объектным литералом в другом месте кода. Вы можете проверить, корректно ли записали деструктурирующее присваивание, попробовав использовать левую сторону с правой стороны другого выражения присваивания:

```
// Начать со структуры данных и сложной деструктуризации
let points = [{x: 1, y: 2}, {x: 3, y: 4}];
let [{x: x1, y: y1}, {x: x2, y: y2}] = points;
// Проверить синтаксис деструктуризации,
// поменяв местами стороны присваивания
let points2=[{x: x1, y: y1},{x: x2, y: y2}]; //points2 == points
```

3.11. Резюме

Ниже перечислены основные моменты, рассмотренные в главе, которые следует запомнить.

- Как записывать и манипулировать числами и строками текста в JavaScript.
- Как работать с остальными элементарными типами JavaScript: булевскими значениями, символами, `null` и `undefined`.
- Отличия между неизменяемыми элементарными типами и изменяемыми ссылочными типами.
- Как JavaScript неявно преобразует значения из одного типа в другой, и каким образом можно делать то же самое явно в своих программах.
- Как объявлять и инициализировать константы и переменные (в том числе с помощью деструктурирующего присваивания) и какова лексическая область видимости объявляемых констант и переменных.

ГЛАВА 4

Выражения и операции

В настоящей главе рассматриваются выражения JavaScript и операции, с помощью которых строятся многие выражения. *Выражение* — это синтаксическая конструкция JavaScript (или фраза), которая может быть *вычислена* для выдачи значения. Константа, литерально встроенная в программу, является очень простым видом выражения. Имя переменной — тоже простое выражение, которое вычисляется в любое значение, присвоенное данной переменной. Сложные выражения строятся из более простых выражений. Например, выражение доступа к элементу массива состоит из одного выражения, результатом которого будет массив, затем открывающей квадратной скобки, еще одного выражения, вычисляемого в целое число, и закрывающей квадратной скобки. Результатом вычисления этого нового, более сложного выражения будет значение, хранящееся по указанному индексу в указанном массиве. Аналогично выражение вызова функции состоит из одного выражения, вычисляемого в объект функции, и нуля или большего количества выражений, которые используются в качестве аргументов функции.

Самый распространенный способ построения сложного выражения из более простых выражений предусматривает применение *операции*. Операция каким-то образом объединяет свои операнды (обычно два) и вычисляет новое значение. Простой пример — операция умножения *. Выражение x * y вычисляется в виде произведения значений выражений x и y. Для простоты иногда мы говорим, что операция *возвращает*, а не "вычисляет" значение.

В главе документируются все операции JavaScript и также объясняются выражения (вроде индексации массивов и вызова функций), которые не используют операции. Если вы знаете еще один язык программирования, имеющий синтаксис в стиле C, то синтаксис большинства выражений и операций JavaScript вам покажется хорошо знакомым.

4.1. Первичные выражения

Простейшими выражениями, известными как *первичные*, считаются автономные выражения — они не включают какие-то более простые выражения.

Первичными выражениями в JavaScript являются константы или *литеральные* значения, определенные ключевые слова языка и ссылки на переменные.

Литералы — это постоянные значения, встроенные напрямую в программу. Вот на что они похожи:

```
1.23          // Числовой литерал
"hello"       // Строковый литерал
/шаблон/      // Литерал в виде регулярного выражения
```

Синтаксис JavaScript для числовых литералов был раскрыт в разделе 3.2. Строковые литералы обсуждались в разделе 3.3. Литералы в виде регулярных выражений были представлены в подразделе 3.3.5 и подробно рассматриваются в разделе 11.3.

Некоторые зарезервированные слова JavaScript являются первичными выражениями:

```
true          // Вычисляется как булевское истинное значение
false         // Вычисляется как булевское ложное значение
null          // Вычисляется как нулевое значение
this          // Вычисляется как "текущий" объект
```

Вы узнали о `true`, `false` и `null` в разделах 3.4 и 3.5. В отличие от остальных ключевое слово `this` — не константа; в разных местах программы оно вычисляется в виде разных значений. Ключевое слово `this` применяется в объектно-ориентированном программировании. Внутри тела метода `this` вычисляется как объект, на котором был вызван метод. За дополнительными сведениями о `this` обращайтесь в раздел 4.5, в главу 8 (особенно в подраздел 8.2.2) и в главу 9.

Наконец, третий тип первичных выражений представляет собой ссылку на переменную, константу или свойство глобального объекта:

```
i             // Вычисляется как значение переменной i
sum           // Вычисляется как значение переменной sum
undefined     // Вычисляется как "неопределенное" свойство
              // глобального объекта
```

Когда любой идентификатор появляется в программе сам по себе, интерпретатор JavaScript предполагает, что это переменная, константа или свойство глобального объекта и ищет его значение. Если переменная с таким именем отсутствует, тогда попытка оценки несуществующей переменной приводит к генерации ошибки ссылки `ReferenceError`.

4.2. Инициализаторы объектов и массивов

Инициализаторы объектов и массивов — это выражения, значениями которых будут вновь созданные объекты или массивы. Такие выражения инициализаторов иногда называются объектными литералами и литералами типа массивов. Однако в отличие от настоящих литералов они не относятся к первичным выражениям, поскольку включают ряд подвыражений, которые указывают значения свойств и элементов. Синтаксис инициализаторов массивов чуть проще и потому мы начнем с них.

Инициализатор массива представляет собой разделяемый запятыми список выражений, содержащийся в квадратных скобках. Значением инициализатора массива является вновь созданный массив. Элементы нового массива инициализируются значениями выражений, разделенных запятыми:

```
[]          // Пустой массив: отсутствие выражений внутри скобок
            // означает, что элементов нет
[1+2,3+4] // Двухэлементный массив. Первый элемент - 3, второй - 7
```

Выражения элементов в инициализаторе массива сами могут быть инициализаторами массивов, т.е. следующие выражения создают вложенные массивы:

```
let matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

Выражения элементов в инициализаторе массива оцениваются каждый раз, когда вычисляется инициализатор массива. Таким образом, при каждом вычислении выражения инициализатора массива его значение может отличаться.

В литерал массива можно включать неопределенные элементы, просто опуская значение между запятыми. Скажем, следующий массив содержит пять элементов, в том числе три неопределенных:

```
let sparseArray = [1,,,,5];
```

После последнего выражения в инициализаторе массива разрешена хвостовая запятая, которая не приводит к созданию неопределенного элемента. Тем не менее, любое выражение доступа к элементу массива для индекса после последнего выражения обязательно даст undefined.

Выражения инициализаторов объектов похожи на выражения инициализаторов массивов, но квадратные скобки заменяются фигурными и каждое подвыражение снабжается префиксом с именем свойства и двоеточием:

```
let p = { x: 2.3, y: -1.2 }; // Объект с двумя свойствами
let q = {};                  // Пустой объект без свойств
q.x = 2.3; q.y = -1.2;       // Теперь q имеет те же свойства, что и p
```

В ES6 объектные литералы обладают гораздо более развитым синтаксисом (детали ищите в разделе 6.10). Объектные литералы могут быть вложенными. Вот пример:

```
let rectangle = {
   upperLeft: { x: 2, y: 2 },
   lowerRight: { x: 4, y: 5 }
};
```

Мы снова столкнемся с инициализаторами объектов и массивов в главах 6 и 7.

4.3. Выражения определений функций

Выражение определения функции определяет функцию JavaScript, а его значением будет вновь определенная функция. В некотором смысле выражение определения функции является "литералом функции" — в том же духе, как инициализатор объекта считается "объектным литералом". Выражение определения функции обычно состоит из ключевого слова function, за которым следует

разделяемый запятыми список из нуля или большего количества идентификаторов (имен параметров) в круглых скобках и блок кода JavaScript (тело функции) в фигурных скобках. Например:

```
// Эта функция возвращает квадрат переданного ей значения.
let square = function(x) { return x * x; };
```

Выражение определения функции также может включать имя для функции. Кроме того, функции могут определяться с использованием оператора, а не выражения функции. В ES6 и последующих версиях выражения функций могут использовать компактный новый синтаксис "стрелочных функций". Полные сведения об определении функций ищите в главе 8.

4.4. Выражения доступа к свойствам

Выражение доступа к свойству вычисляется как значение свойства объекта или элемент массива. В JavaScript определены два вида синтаксиса для доступа к свойствам:

```
выражение . идентификатор
выражение [ выражение ]
```

Первый стиль доступа к свойствам представляет собой выражение, за которым следуют точка и идентификатор. Выражение указывает объект, а идентификатор — имя желаемого свойства. Во втором стиле доступа к свойствам за первым выражением (объект или массив) идет еще одно выражение в квадратных скобках. Во втором выражении указывается имя желаемого свойства либо индекс желаемого элемента массива. Ниже приведено несколько конкретных примеров:

```
let o = {x: 1, y: {z: 3}};  // Пример объекта
let a = [o, 4, [5, 6]];     // Пример массива, содержащего объект
o.x                         // => 1: свойство x выражения o
o.y.z                       // => 3: свойство z выражения o.y
o["x"]                      // => 1: свойство x of object o
a[1]                        // => 4: элемент по индексу 1 выражения a
a[2]["1"]                   // => 6: элемент по индексу 1 выражения a[2]
a[0].x                      // => 1: свойство x выражения a[0]
```

В обоих типах выражения доступа к свойству первым вычисляется выражение перед . или [. Если значением оказывается `null` или `undefined`, тогда выражение генерирует ошибку типа `TypeError`, т.к. это два значения JavaScript, которые не могут иметь свойства. Если за выражением объекта следует точка и идентификатор, то выполняется поиск значения свойства с именем, указанным в идентификаторе, и оно становится общим значением выражения. Если за выражением объекта следует еще одно выражение в квадратных скобках, тогда второе выражение вычисляется и преобразуется в строку. Общим значением выражения будет значение свойства с именем, соответствующим данной строке. В обоих случаях, если именованное свойство не существует, тогда значением выражения доступа к свойству становится `undefined`.

Из двух вариантов доступа к свойству синтаксис *.идентификатор* проще, но важно помнить, что он может применяться, только когда желаемое свойство имеет имя, которое считается допустимым идентификатором, и когда при написании программы имя свойства известно. Если имя свойства включает пробелы или знаки пунктуации, либо оно представляет собой число (для массивов), тогда придется использовать вариант с квадратными скобками. Квадратные скобки также применяются в случае, когда имя свойства является не статическим, а результатом какого-то вычисления (пример ищите в подразделе 6.3.1).

Объекты и их свойства детально раскрываются в главе 6, а массивы и их элементы — в главе 7.

4.4.1. Условный доступ к свойствам

В ES2020 появились два новых вида выражений доступа к свойствам:

выражение `?.` *идентификатор*
выражение `?.[` *выражение* `]`

В JavaScript не располагают свойствами только два значения, `null` и `undefined`. В обыкновенном выражении доступа к свойствам, использующем `.` или `[]`, вы получите ошибку `TypeError`, если результатом вычисления выражения слева оказывается `null` или `undefined`. Чтобы защититься от ошибок такого типа, вы можете применять синтаксис `?.` и `?.[]`.

Давайте обсудим выражение `a?.b`. Если `a` равно `null` или `undefined`, тогда выражение вычисляется как `undefined` без попытки доступа к свойству `b`. Если `a` имеет какое-то другое значение, то результатом вычисления `a?.b` становится то, что получилось бы при вычислении `a.b` (и если `a` не имеет свойства по имени `b`, тогда значением снова будет `undefined`).

Показанную форму выражения доступа к свойствам иногда называют "необязательным связыванием", поскольку она также работает для более длинных "связанных" выражений доступа к свойствам, например:

```
let a = { b: null };
a.b?.c.d    // => undefined
```

`a` — объект, так что `a.b` — допустимое выражение доступа к свойству. Но значением `a.b` является `null`, и потому `a.b.c` сгенерировало бы ошибку `TypeError`. За счет использования `?.` вместо `.` мы избегаем ошибки и `a.b?.c` вычисляется как `undefined`. Это означает, что `(a.b?.c).d` сгенерирует ошибку `TypeError`, т.к. выражение попытается обратиться к свойству значения `undefined`. Но — и это очень важный аспект "необязательного связывания" — `a.b?.c.d` (без круглых скобок) просто вычисляется как `undefined` и не приводит к генерации ошибки. Причина в том, что доступ к свойствам с помощью `?.` действует по принципу "короткого замыкания": если подвыражение слева от `?.` вычисляется как `null` или `undefined`, тогда целое выражение немедленно вычисляется как `undefined` безо всяких дальнейших попыток доступа к свойствам.

Разумеется, если a.b — объект, который не имеет свойства по имени c, то a.b?.c.d снова сгенерирует ошибку TypeError, и мы захотим применить еще один условный доступ к свойству:

```
let a = { b: {} };
a.b?.c?.d          // => undefined
```

Условный доступ к свойствам также возможен с использованием ?.[] вместо []. Если в выражении a?.[b][c] значением a оказывается null или undefined, то целое выражение немедленно вычисляется как undefined, а подвыражения b и c даже не оцениваются. Если любое из этих подвыражений имеет побочные эффекты, тогда они не произойдут, когда переменная a не определена:

```
let a;             // Ой, мы забыли инициализировать эту переменную!
let index = 0;
try {
   a[index++];     // Генерируется TypeError
} catch(e) {
   index           // => 1: инкрементирование происходит перед
                   // генерацией TypeError
}
a?.[index++]       //=>undefined:поскольку переменная a не определена
index              //=>1: переменная index не инкрементировалась
                   // из-за короткого замыкания ?.[]
a[index++]         // !TypeError: undefined индексировать нельзя
```

Условный доступ к свойствам посредством ?. и ?.[] — одна из новейших возможностей JavaScript. По состоянию на начало 2020 года такой новый синтаксис поддерживался в текущих или бета-версиях большинства ведущих браузеров.

4.5. Выражения вызова

Выражение вызова — это синтаксис JavaScript для вызова (либо выполнения) функции или метода. Оно начинается с выражения функции, которое идентифицирует функцию, подлежащую вызову. За выражением функции следуют открывающая круглая скобка, разделяемый запятыми список из нуля или большего количества выражений аргументов и закрывающая круглая скобка. Вот несколько примеров:

```
f(0)               // f - выражение функции; 0 - выражение аргумента
Math.max(x,y,z)    // Math.max - функция; x, y и z - аргументы
a.sort()           // a.sort - функция; аргументы отсутствуют
```

При вычислении выражения вызова первым вычисляется выражение функции, после чего вычисляются выражения аргументов, чтобы получить список значений аргументов. Если значение выражения функции не является функцией, тогда генерируется ошибка TypeError. Далее значения аргументов по порядку присваиваются именам параметров, указанным во время определения функции, и выполняется тело функции. Если в функции применяется оператор return

для возврата значения, то оно становится значением выражения вызова, иначе значением выражения вызова будет undefined. Исчерпывающие сведения о вызове функций, включая объяснение того, что происходит, когда количество выражений аргументов не совпадает с количеством параметров в определении функции, приведены в главе 8.

Каждое выражение вызова содержит пару круглых скобок и выражение перед открывающей круглой скобкой. Если оно представляет собой выражения доступа к свойству, тогда мы получаем *вызов метода*. В вызовах методов объект или массив, к свойству которого происходит доступ, становится значением ключевого слова this, пока выполняется тело функции. В итоге обеспечивается парадигма объектно-ориентированного программирования, когда функции (называемые "методами" при таком способе использования) работают на объекте, которому они принадлежат. Детали ищите в главе 9.

4.5.1. Условный вызов

В ES2020 функцию можно вызывать также с применением `?.()` вместо `()`. Обычно при вызове функции в случае, если выражение слева от круглых скобок оказывается null или undefined либо чем-то, отличающимся от функции, то генерируется ошибка TypeError. В контексте нового синтаксиса вызовов `?.()`, если выражение слева от `?.` вычисляется как null и undefined, тогда результатом вычисления целого выражения вызова будет undefined без генерирования каких-либо исключений.

Объекты массивов имеют метод `sort()`, которому дополнительно можно передать аргумент функции, устанавливающий желаемый порядок сортировки элементов массива. До версии ES2020 для написания метода вроде `sort()`, который принимает необязательный аргумент функции, как правило, вам пришлось бы использовать оператор if, чтобы проверять, определен ли этот аргумент функции, прежде чем вызывать его в теле if:

```
function square(x, log) { // Второй аргумент - необязательная функция
   if (log) {             // Если необязательная функция передана,
      log(x);             // тогда вызвать ее
   }
   return x * x;          // Возвратить квадрат аргумента
}
```

Однако с помощью синтаксиса условных вызовов ES2020 вы можете просто написать вызов функции с применением `?.()`, зная о том, что вызов произойдет только в том случае, если на самом деле имеется значение, которое должно вызываться:

```
function square(x, log) {  // Второй аргумент - необязательная функция
   log?.(x);               // Вызвать функцию, если она есть
   return x * x;           // Возвратить квадрат аргумента
}
```

Тем не менее, обратите внимание, что синтаксис `?.()` только проверяет, равна ли левая сторона null или undefined. Он не выясняет, действительно ли

значение является функцией. Таким образом, функция `square()` в показанном примере по-прежнему сгенерирует исключение, если вы передадите ей, скажем, два числа.

Подобно выражениям условного доступа к свойствам (см. подраздел 4.4.1) вызов функции посредством `?.()` действует по принципу "короткого замыкания": если значение слева от `?.` равно `null` или `undefined`, тогда выражения аргументов внутри круглых скобок вычисляться не будут:

```
let f = null, x = 0;
try {
   f(x++);     // Генерируется TypeError, потому что f равно null
} catch(e) {
   x           // => 1: x инкрементируется перед генерацией исключения
}
f?.(x++)       // => undefined: f равно null, но никакие исключения
               //                не генерируются
x              // => 1: инкрементирование пропускается из-за
               //       короткого замыкания
```

Выражения условных вызовов с `?.()` в той же степени хорошо работают для методов, как они делают это для функций. Но поскольку вызов метода также предусматривает доступ к свойству, стоит потратить некоторое время, чтобы убедиться в том, что вы понимаете отличия между следующими выражениями:

```
o.m()          // Обыкновенный доступ к свойству, обыкновенный вызов
o?.m()         // Условный доступ к свойству, обыкновенный вызов
o.m?.()        // Обыкновенный доступ к свойству, условный вызов
```

В первом выражении переменная `o` должна быть объектом со свойством `m`, а значение данного свойства обязано быть функцией. Во втором выражении, если `o` равно `null` или `undefined`, то выражение вычисляется как `undefined`. Но если переменная `o` имеет любое другое значение, тогда она должна располагать свойством `m`, значение которого является функцией. В третьем выражении переменная `o` не должна быть `null` или `undefined`. Если она не имеет свойства `m` или значение `m` равно `null`, то полное выражение вычисляется как `undefined`.

Условный вызов с помощью `?.()` — одна из новейших возможностей JavaScript. По состоянию на начало 2020 года такой новый синтаксис поддерживался в текущих или бета-версиях большинства ведущих браузеров.

4.6. Выражения создания объектов

Выражение создания объекта создает новый объект и вызывает функцию (называемую конструктором) для инициализации свойств этого объекта. Выражения создания объектов похожи на выражения вызовов, но предваряются ключевым словом `new`:

```
new Object()
new Point(2,3)
```

Если функции конструктора в выражении создания объекта аргументы не передаются, тогда пустую пару круглых скобок можно опустить:

```
new Object
new Date
```

Значением выражения создания объекта будет вновь созданный объект. Конструкторы более подробно обсуждаются в главе 9.

4.7. Обзор операций

Операции используются в JavaScript для арифметических выражений, выражений сравнения, логических выражений, выражений присваивания и т.д. В табл. 4.1 приведена сводка по операциям, которая послужит удобной ссылкой.

Таблица 4.1. Операции JavaScript

Операция	Действие	Ассоциативность	Количество операндов	Типы
`++`	Префиксный и постфиксный инкремент	Справа налево	1	тип левого значения → числовой
`--`	Префиксный и постфиксный декремент	Справа налево	1	тип левого значения → числовой
`-`	Отрицание числа	Справа налево	1	числовой → числовой
`+`	Преобразование в число	Справа налево	1	любой → числовой
`~`	Инвертирование битов	Справа налево	1	целочисленный → целочисленный
`!`	Инвертирование булевского значения	Справа налево	1	булевский → булевский
`delete`	Удаление свойства	Справа налево	1	тип левого значения → булевский
`typeof`	Установление типа операнда	Справа налево	1	любой → строковый
`void`	Возвращение значения `undefined`	Справа налево	1	любой → `undefined`
`**`	Возведение в степень	Справа налево	2	числовой, числовой → числовой
`*`, `/`, `%`	Умножение, деление, остаток от деления	Слева направо	2	число́вой, числовой → числовой

Операция	Действие	Ассоциативность	Количество операндов	Типы
+, -	Сложение, вычитание	Слева направо	2	числовой, числовой → числовой
+	Конкатенация строк	Слева направо	2	строковый, строковый → строковый
<<	Сдвиг влево	Слева направо	2	целочисленный, целочисленный → целочисленный
>>	Сдвиг вправо с расширением знака	Слева направо	2	целочисленный, целочисленный → целочисленный
>>>	Сдвиг вправо с дополнением нулями	Слева направо	2	целочисленный, целочисленный → целочисленный
<, <=, >, >=	Сравнение в числовом порядке	Слева направо	2	числовой, числовой → булевский
<, <=, >, >=	Сравнение в алфавитном порядке	Слева направо	2	строковый, строковый → булевский
instanceof	Проверка класса объекта	Слева направо	2	объектный, функциональный → булевский
in	Проверка, существует ли свойство	Слева направо	2	любой, объектный → булевский
==	Проверка на предмет нестрогого равенства	Слева направо	2	любой, любой → булевский
!=	Проверка на предмет нестрогого неравенства	Слева направо	2	любой, любой → булевский
===	Проверка на предмет строгого равенства	Слева направо	2	любой, любой → булевский
!==	Проверка на предмет строгого неравенства	Слева направо	2	любой, любой → булевский
&	Вычисление побитового И	Слева направо	2	целочисленный, целочисленный → целочисленный
^	Вычисление побитового исключающего ИЛИ	Слева направо	2	целочисленный, целочисленный → целочисленный
|	Вычисление побитового ИЛИ	Слева направо	2	целочисленный, целочисленный → целочисленный
&&	Вычисление логического И	Слева направо	2	любой, любой → любой
||	Вычисление логического ИЛИ	Слева направо	2	любой, любой → любой

Операция	Действие	Ассоциативность	Количество операндов	Типы
`??`	Выбор первого определенного операнда	Слева направо	2	любой, любой → любой
`?:`	Выбор второго или третьего операнда	Справа налево	3	булевский, любой, любой → любой
`=`	Присваивание переменной или свойства	Справа налево	2	тип левого значения, любой → любой
`**=, *=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=`	Оперирование и присваивание	Справа налево	2	тип левого значения, любой → любой
`,`	Отбрасывание первого операнда, возвращение второго	Слева направо	2	любой, любой → любой

Обратите внимание, что большинство операций представлены символами пунктуации, такими как + и =. Однако некоторые операции представлены ключевыми словами наподобие `delete` и `instanceof`. Операции, представленные ключевыми словами, являются обыкновенными операциями, как те, что выражены с помощью символов пунктуации; у них просто менее лаконичный синтаксис.

Информация в табл. 4.1 организована по приоритетам операций. Операции, перечисленные первыми, имеют более высокий приоритет, чем операции, идущие последними. Операции, разделенные горизонтальной линией, имеют разные уровни приоритета. В колонке "Ассоциативность" указана ассоциативность операций — слева направо или справа налево. В колонке "Количество операндов" приводится количество операндов. В колонке "Типы" перечисляются ожидаемые типы операндов и (после символа →) тип результата операции. В последующих подразделах объясняются концепции приоритета, ассоциативности и типа операндов. Затем будут описаны сами операции.

4.7.1. Количество операндов

Операции можно разбить на категории, основываясь на количестве ожидаемых ими операндов (их *арности*). Большинство операций JavaScript, подобных операции умножения *, являются *бинарными операциями*, которые объединяют два выражения в одно более сложное выражение. То есть они ожидают два операнда. В JavaScript также поддерживается несколько *унарных операций*, которые превращают одиночное выражение в одиночное более сложное выражение. Операция - в выражении -x представляет собой унарную операцию, выполня-

ющую отрицание операнда x. Наконец, в JavaScript имеется одна *тернарная операция* — условная операция ?:, которая объединяет три выражения в одно.

4.7.2. Типы операндов и результата

Некоторые операции работают со значениями любого типа, но большинство ожидают, что их операнды относятся к специфическим типам, и возвращают (или вычисляют) значение специфического типа. В колонке "Типы" в табл. 4.1 указаны типы операндов (перед стрелкой) и тип результата (после стрелки) для операций.

Операции JavaScript обычно по мере надобности преобразуют типы (см. раздел 3.9) своих операндов. Операция умножения * ожидает числовые операнды, но выражение "3" * "5" допустимо, потому что интерпретатор JavaScript может преобразовать операнды в числа. Значением такого выражения, конечно же, будет число 15, не строка "15". Вспомните также, что каждое значение JavaScript является либо "истиной", либо "ложью", поэтому операции, которые ожидают булевские операнды, будут работать с операндами любого типа.

Определенные операции ведут себя по-разному в зависимости от типов своих операндов. В частности, операция + складывает числовые операнды, но производит конкатенацию строковых операндов. Аналогично операции сравнения, такие как <, выполняют сравнение в числовом или алфавитном порядке в зависимости от типов операндов. В описаниях индивидуальных операций объясняются их зависимости от типов и указываются преобразования типов, которые они делают.

Обратите внимание, что операции присваивания и ряд других операций, перечисленных в табл. 4.1, ожидают операнд типа левого значения. *Левое значение* (lvalue) — исторический термин, означающий "выражение, которое может законно находится с левой стороны операции присваивания". В JavaScript левыми значениями являются переменные, свойства объектов и элементы массивов.

4.7.3. Побочные эффекты операций

Вычисление простого выражения вроде 2 * 3 никогда не влияет на состояние вашей программы и любые будущие вычисления, выполняемые программой, не будут затронуты упомянутым вычислением. Тем не менее, некоторые выражения имеют побочные эффекты, и их вычисление может влиять на результат будущих вычислений. Самым очевидным примером служат операции присваивания: если вы присвоите значение переменной или свойству, то это изменит значение любого выражения, в котором задействована данная переменная или свойство. Похожая ситуация с операциями инкремента ++ и декремента --, т.к. они выполняются неявное присваивание. Операция delete тоже имеет побочные эффекты: удаление свойства подобно (но не совпадает) присваиванию значения undefined свойству.

С остальными операциями JavaScript не связаны побочные эффекты, но выражения вызовов функций и создания объектов будут иметь побочные эффек-

ты, если в теле функции или конструктора применяется какая-то операция с побочными эффектами.

4.7.4. Приоритеты операций

Операции в табл. 4.1 перечислены в порядке от высокого приоритета до низкого и с помощью горизонтальных линий отделяются группы операций с одним и тем же уровнем приоритета. Приоритеты операций управляют порядком, в котором операции выполняются. Операции с более высоким приоритетом (ближе к началу таблицы) выполняются перед операциями с более низким приоритетом (ближе к концу таблицы).

Рассмотрим следующее выражение:

```
w = x + y*z;
```

Операция умножения * имеет более высокий приоритет, чем операция сложения +, поэтому умножение выполняется раньше сложения. Операция присваивания = имеет самый низкий приоритет, так что присваивание выполняется после того, как все операции в правой стороне завершены.

Приоритеты операций можно переопределять за счет явного использования круглых скобок. Чтобы заставить сложение в предыдущем примере выполниться первым, запишите так:

```
w = (x + y)*z;
```

Обратите внимание, что выражения доступа к свойствам и вызова имеют более высокий приоритет, нежели любая операция, перечисленная в табл. 4.1. Взгляните на приведенное ниже выражение:

```
// my - объект со свойством по имени functions, значением которого
// является массив функций. Мы вызываем функцию номер x, передаем
// ей аргумент y и запрашиваем тип возвращенного значения.
typeof my.functions[x](y)
```

Хотя `typeof` — одна из операций с самым высоким приоритетом, она будет выполняться на результате доступа к свойству, индексации массива и вызове функции, которые обладают более высоким приоритетом, чем `typeof`.

На практике, если вы не совсем уверены в приоритетах своих операций, то проще всего применять круглые скобки, делая порядок вычисления явным. Важно знать следующие правила: умножение и деление выполняются перед сложением и вычитанием, а присваивание имеет очень низкий приоритет и почти во всех случаях выполняется последним.

Когда в JavaScript добавляются новые операции, они не всегда естественным образом вписываются в эту схему приоритетов. Операция `??` (см. подраздел 4.13.2) показана в табл. 4.1 как имеющая более низкий приоритет, чем `||` и `&&`, но на самом деле ее приоритет относительно упомянутых операций не определен, и стандарт ES2020 требует явного использования круглых скобок при смешивании `??` с `||` или `&&`. Аналогично новая операция возведения в степень `**` не обладает четко определенным приоритетом относительно унарной операции

отрицания и потому при комбинировании отрицания и возведения в степень придется применять круглые скобки.

4.7.5. Ассоциативность операций

В табл. 4.1 для каждой операции указана ее *ассоциативность*, которая определяет порядок выполнения операций с одинаковыми приоритетами. Ассоциативность слева направо означает, что операции выполняются слава направо. Например, операция вычитания имеет ассоциативность слева направо, поэтому:

```
w = x - y - z;
```

то же самое, что и:

```
w = ((x - y) - z);
```

С другой стороны, следующие выражения:

```
y = a ** b ** c;
x = ~-y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

эквивалентны таким выражениям:

```
y = (a ** (b ** c));
x = ~(-y);
w = (x = (y = z));
q = a?b:(c?d:(e?f:g));
```

потому что операция возведения в степень, унарная операция, операция присваивания и тернарная условная операция имеют ассоциативность справа налево.

4.7.6. Порядок вычисления

Приоритеты и ассоциативность операций задают порядок их выполнения в сложном выражении, но они не указывают порядок, в котором вычисляются подвыражения. Интерпретатор JavaScript всегда вычисляет выражения строго в порядке слева направо. Например, в выражении `w = x + y * z` подвыражение `w` вычисляется первым, а за ним `x`, `y` и `z`. Далее значения `y` и `z` перемножаются, произведение складывается со значением `x` и результат присваивается переменной или свойству, указанному выражением `w`. Добавление круглых скобок к выражениям может изменить относительный порядок выполнения умножения, сложения и присваивания, но не порядок вычисления слева направо.

Порядок вычисления следует учитывать только в ситуации, когда какое-то вычисляемое выражение имеет побочные эффекты, которые влияют на значение другого выражения. Если выражение `x` инкрементирует переменную, используемую выражением `z`, тогда тот факт, что `x` вычисляется перед `z`, важен.

4.8. Арифметические выражения

В этом разделе рассматриваются операции, которые выполняют арифметические и другие числовые действия над своими операндами. Операции возведения в степень, умножения, деления и вычитания прямолинейны и раскрываются первыми. Для операции сложения выделен собственный подраздел, поскольку она также может выполнять конкатенацию строк, и вдобавок с ней связано несколько необычных правил преобразования типов. Унарные и побитовые операции тоже описаны в отдельных подразделах.

Большинство арифметических операций (за исключением отмеченных ниже) можно применять с операндами `BigInt` (см. подраздел 3.2.5) или с обыкновенными числами при условии, что эти два типа не смешиваются.

Базовыми арифметическими операциями являются `**` (возведение в степень), `*` (умножение), `/` (деление), `%` (модуль: остаток от деления), `+` (сложение) и `-` (вычитание). Как упоминалось, мы обсудим операцию `+` в отдельном разделе. Остальные пять базовых операций просто оценивают свои операнды, при необходимости преобразуют значения в числа и затем вычисляют степень, произведение, частное, остаток или разность. Нечисловые операнды, которые не удается преобразовать в числа, получают значение `NaN`. Если один из двух операндов равен (или преобразован в) `NaN`, тогда результатом операции будет (почти всегда) `NaN`.

Операция `**` имеет более высокий приоритет, чем операции `*`, `/` и `%` (приоритет которых в свою очередь выше, чем у `+` и `-`). В отличие от других операций операция `**` работает справа налево, так что `2**2**3` — то же самое, что и `2**8`, но не `4**3`. В выражениях наподобие `-3**2` присутствует естественная неоднозначность. В зависимости от относительного приоритета унарного минуса и возведения в степень указанное выражение может означать `(-3)**2` или `-(3**2)`. Разные языки обрабатывают это по-разному, и вместо того, чтобы принять какую-то сторону, интерпретатор JavaScript просто сообщает о синтаксической ошибке в случае отсутствия круглых скобок в данном случае, вынуждая вас записывать недвусмысленное выражение. Операция `**` в JavaScript относится к наиболее новым: она была добавлена в язык с выходом ES2016. Однако с самых ранних версий JavaScript доступна функция `Math.pow()`, которая выполняет то же действие, что и операция `**`.

Операция `/` делит свой первый операнд на второй. Если вы привыкли к языкам программирования, в которых проводится различие между целыми числами и числами с плавающей точкой, то могли бы ожидать получения целочисленного результата при делении одного целого на другое. Тем не менее, в JavaScript все числа представлены с плавающей точкой, поэтому все операции деления производят результаты с плавающей точкой: результатом вычисления `5/2` будет `2.5`, а не `2`. Деление на ноль выдает положительную или отрицательную бесконечность, в то время как `0/0` оценивается в `NaN`: ни один из подобных случаев не генерирует ошибку.

Операция `%` делит по модулю первый операнд на второй. Другими словами, она возвращает остаток от целочисленного деления первого операнда на второй.

Знак результата будет совпадать со знаком первого операнда. Скажем, результат вычисления `5 % 2` равен 1, а результат `-5 % 2` составляет `-1`.

Наряду с тем, что операция деления по модулю обычно используется с целочисленными операндами, она также работает со значениями с плавающей точкой. Например, результат `6.5 % 2.1` равен `0.2`.

4.8.1. Операция +

Бинарная операция + выполняет сложение числовых операндов или конкатенацию строковых операндов:

```
1 + 2                       // => 3
"hello" + " " + "there"     // => "hello there"
"1" + "2"                   // => "12"
```

Когда значениями обоих операндов являются либо числа, либо строки, тогда то, что делает операция +, вполне очевидно. Однако в любом другом случае требуется преобразование типов, и то, какое действие выполнится, зависит от выполненного преобразования. Правила преобразования для операции + отдают предпочтение конкатенации строк: если один из операндов оказывается строкой или объектом, преобразуемым в строку, то другой операнд преобразуется в строку и выполняется конкатенация. Сложение выполняется лишь тогда, когда ни один из операндов не подобен строке.

Формально операция + ведет себя так, как описано ниже.

- Если любое из значений двух операндов является объектом, тогда он преобразуется в элементарное значение с применением алгоритма, описанного в подразделе 3.9.3. Объекты `Date` преобразуются посредством своего метода `toString()`, а все остальные объекты преобразуются через `valueOf()`, если этот метод возвращает элементарное значение. Тем не менее, большинство объектов не располагает полезным методом `valueOf()`, так что они преобразуются тоже через `toString()`.
- После преобразования объекта в элементарное значение, если любой из двух операндов оказывается строкой, то другой преобразуется в строку и выполняется конкатенация.
- В противном случае оба операнда преобразуются в числа (или в `NaN`) и выполняется сложение.

Ниже приведено несколько примеров:

```
1 + 2          // => 3: сложение
"1" + "2"      // => "12": конкатенация
"1" + 2        //=>"12": конкатенация после преобразования числа в строку
1 + {}         // => "1[object Object]": конкатенация после
               //                        преобразования объекта в строку
true + true    // => 2: сложение после преобразования
               //       булевского значения в число
2 + null       // => 2: сложение после преобразования null в 0
2 + undefined  // => NaN: сложение после преобразования undefined в NaN
```

В заключение важно отметить, что когда операция + используется со строками и числами, то она может не быть ассоциативной, т.е. результат будет зависеть от порядка выполнения действий.

Вот пример:

```
1 + 2 + " blind mice"       // => "3 blind mice"
1 + (2 + " blind mice")     // => "12 blind mice"
```

В первой строке кода круглые скобки отсутствуют, а операция + имеет ассоциативность слева направо, поэтому сначала складываются два числа и выполняется конкатенация их суммы со строкой. Во второй строке кода круглые скобки изменяют порядок действий: происходит конкатенация числа 2 со строкой для получения новой строки. Затем выполняется конкатенация числа 1 с новой строкой для получения финального результата.

4.8.2. Унарные арифметические операции

Унарные операции модифицируют значение единственного операнда, чтобы получить новое значение. Все унарные операции в JavaScript имеют высокий приоритет и являются ассоциативными справа налево. Унарные арифметические операции, описанные в настоящем разделе (+, –, ++ и --), при необходимости преобразуют свой единственный операнд в число. Обратите внимание, что символы пунктуации + и — применяются в качестве и унарных, и бинарных операций.

Ниже перечислены унарные арифметические операции.

Унарный плюс **(+)**

Операция унарного плюса преобразует свой операнд в число (или в `NaN`) и возвращает преобразованное значение. В случае использования с операндом, который уже является числом, она ничего не делает. Операцию унарного плюса нельзя применять со значениями `BigInt`, т.к. они не могут быть преобразованы в обыкновенные числа.

Унарный минус **(-)**

Когда — используется как унарная операция, операнд при необходимости преобразуется в число и знак результата изменяется.

Инкремент **(++)**

Операция ++ инкрементирует свой единственный операнд (т.е. добавляет к нему 1), который обязан быть левым значением (переменной, элементом массива или свойством объекта). Операция преобразует свой операнд в число, добавляет к нему 1 и присваивает инкрементированное значение обратно переменной, элементу или свойству.

Возвращаемое значение операции ++ зависит от ее позиции относительно операнда. Когда операция ++ находится перед операндом, она называется префиксным инкрементом, инкрементирует операнд и результатом вычисления будет инкрементированное значение операнда. Когда операция

++ находится после операнда, она называется постфиксным инкрементом, инкрементирует операнд и результатом вычисления будет значение операнда *до инкрементирования*. Взгляните на отличие между следующими двумя строками кода:

```
let i = 1, j = ++i;    // i и j равны 2
let n = 1, m = n++;    // n равно 2, m равно 1
```

Обратите внимание, что выражение x++ не всегда будет тем же, что и x=x+1. Операция ++ никогда не выполняет конкатенацию строк: она всегда преобразует свой операнд в число и инкрементирует его. Если x — строка "1", то ++x будет числом 2, но x+1 — строкой "11".

Кроме того, из-за автоматической вставки точек с запятой интерпретатором JavaScript вы не можете размещать разрыв строки между операцией постфиксного инкремента и операндом, который ей предшествует. Если вы поступите так, тогда интерпретатор JavaScript будет трактовать операнд как полный оператор и вставит точку запятой перед операцией.

Операция инкремента в своих двух формах, префиксной и постфиксной, чаще всего применяется для инкрементирования счетчика, управляющего циклом `for` (см. подраздел 5.4.3).

***Декремент* (--)**

Операция -- ожидает операнда, являющегося левым значением. Она преобразует значение операнда в число, вычитает из него 1 и присваивает декрементированное значение обратно операнду. Подобно ++ возвращаемое значение операции -- зависит от ее позиции относительно операнда. Когда она используется перед операндом, то декрементирует и возвращает декрементированное значение, а когда после операнда, то декрементирует операнд, но возвращает значение *до декрементирования*. В случае применения после операнда разрыв строки между операндом и операцией не разрешен.

4.8.3. Побитовые операции

Побитовые операции выполняют низкоуровневые манипуляции битами в двоичном представлении чисел. Хотя они не совершают традиционные арифметические действия, но относятся здесь к категории арифметических операций, потому что работают с числовыми операндами и возвращают числовое значение. Четыре операции выполняют действия булевой алгебры над индивидуальными битами операндов и ведут себя так, как если бы каждый бит в каждом операнде был булевским значением (1 = истина, 0 = ложь). Другие три побитовых операции используются для сдвига битов влево и вправо. Такие операции обычно не применяются в программировании на JavaScript, и если вы не знакомы с двоичным представлением целых чисел, включая два представления отрицательных чисел в дополнительном коде, то вполне можете пропустить текущий раздел.

Побитовые операции ожидают целочисленных операндов и ведут себя так, будто их значения представлены в виде 32-битных целых чисел, а не 64-битных значений с плавающей точкой. Эти операции при необходимости преобразуют свои операнды в числа и затем приводят числовые значения к 32-битным целым, отбрасывая дробные части и биты после 32-го. Операции сдвига требуют с правой стороны операнда между 0 и 31. После его преобразования в 32-битное целое без знака они отбрасывают все биты после 5-го, получая число в подходящем диапазоне. Удивительно, но `NaN`, `Infinity` и `-Infinity` преобразуются в 0, когда используются в качестве операндов этих побитовых операций.

Все побитовые операции кроме `>>>` могут применяться с обыкновенными числовыми операндами или с операндами `BigInt` (см. подраздел 3.2.5).

***Побитовое И* (`&`)**

Операция `&` выполняет булевскую операцию И с каждым битом своих целочисленных аргументов. Бит устанавливается в результате, только если соответствующий бит установлен в обоих операндах. Например, `0x1234 & 0x00FF` вычисляется как `0x0034`.

***Побитовое ИЛИ* (`|`)**

Операция `|` выполняет булевскую операцию ИЛИ с каждым битом своих целочисленных аргументов. Бит устанавливается в результате, если соответствующий бит установлен в одном или обоих операндах. Например, `0x1234 | 0x00FF` вычисляется как `0x12FF`.

***Побитовое исключающее ИЛИ* (`^`)**

Операция `^` выполняет булевскую операцию исключающего ИЛИ с каждым битом своих целочисленных аргументов. Исключающее ИЛИ означает, что либо первый операнд равен `true`, либо второй операнд равен `true`, но не оба. Бит устанавливается в результате этой операции, если соответствующий бит установлен в одном из двух операндов (но не в обоих). Например, `0xFF00 ^ 0xF0F0` вычисляется как `0x0FF0`.

***Побитовое НЕ* (`~`)**

Операция `~` является унарной операцией, указываемой перед своим единственным целочисленным операндом. Она работает путем переключения всех битов в операнде. Из-за способа представления целых чисел со знаком в JavaScript применение операции `~` к значению эквивалентно изменению его знака и вычитанию 1. Например, `~0x0F` вычисляется как `0xFFFFFFF0`, или –16.

***Сдвиг влево* (`<<`)**

Операция `<<` перемещает все биты в первом операнде влево на количество позиций, указанное во втором операнде, которое должно быть целым числом между 0 и 31. Скажем, в действии `a << 1` первый бит `a` становится вторым, второй бит `a` — третьим и т.д. Для нового первого бита используется ноль, а значение 32-го бита утрачивается. Сдвиг значения влево на одну позицию эквивалентен умножению на 2, сдвиг на две позиции — умножению на 4 и т.д. Например, `7 << 2` вычисляется как 28.

Сдвиг вправо с расширением знака (>>)

Операция >> перемещает все биты в первом операнде вправо на количество позиций, указанное во втором операнде (целое число между 0 и 31). Биты, сдвинутые за правую границу числа, утрачиваются. Биты, заполняемые слева, зависят от знакового бита исходного операнда, чтобы предохранять знак результата. Если первый операнд положительный, то старшие биты результата заполняются нулями; если первый операнд отрицательный, тогда в старшие биты результата помещаются единицы. Сдвиг положительного значения вправо на одну позицию эквивалентен делению на 2 (с отбрасыванием остатка), сдвиг вправо на две позиции равноценен целочисленному делению на 4 и т.д. Например, `7 >> 1` вычисляется как 3, но имейте в виду, что `-7 >> 1` вычисляется как –4.

Сдвиг вправо с дополнением нулями (>>>)

Операция >>> похожа на операцию >> за исключением того, что старшие биты при сдвиге будут всегда нулевыми независимо от знака первого операнда. Операция полезна, когда 32-битные значения желательно трактовать как целые числа без знака. Например, `-1 >> 4` вычисляется как –1, но `-1 >>> 4` — как `0x0FFFFFFF`. Это единственная побитовая операция JavaScript, которую нельзя применять со значениями `BigInt`. Тип `BigInt` не представляет отрицательные числа, устанавливая старший бит так, как поступают 32-битные целые числа, а потому данная операция является просто дополнением предшествующих двух.

4.9. Выражения отношений

В этом разделе описаны операции отношений JavaScript, которые проверяют отношение (вроде "равно", "меньше чем" или "является свойством") между двумя значениями и возвращают `true` или `false` в зависимости от того, существует ли такое отношение. Результатом вычисления выражений отношений всегда будет булевское значение, которое часто используется для управления потоком выполнения программы в операторах `if`, `while` и `for` (см. главу 5). В последующих подразделах документируются операции равенства и неравенства, операции сравнения и еще две операции отношений JavaScript, `in` и `instanceof`.

4.9.1. Операции равенства и неравенства

Операции == и === проверяют, одинаковы ли два значения, с применением двух разных определений тождественности. Обе операции принимают операнды любого типа и обе возвращают `true`, если их операнды одинаковы, и `false`, если они отличаются. Операция === известна как операция строгого равенства (или временами операция идентичности) и проверяет, "идентичны" ли два ее операнда, используя строгое определение тождественности. Операция == известна как операция равенства; она проверяет, "равны" ли два ее операнда, с применением более мягкого определения тождественности, которое допускает преобразования типов.

Операции != и !== проверяют прямую противоположность операциям == и ===. Операция неравенства != возвращает false, если два значения равны друг другу согласно операции ==, и true в противном случае. Операция !== возвращает false, если два значения строго равны друг другу, и true в противном случае. Как будет показано в разделе 4.10, операция ! выполняет булевскую операцию НЕ. В итоге легче запомнить, что != и !== означают "не равно" и "не строго равно".

Операции =, == и ===

В JavaScript поддерживаются операции =, == и ===. Удостоверьтесь в том, что понимаете отличия между этими операциями присваивания, равенства и строгого равенства, и во время написания кода будьте внимательны при выборе подходящей операции! Хотя заманчиво читать все три операции просто как "равно", во избежание путаницы имеет смысл трактовать операцию = как "получает" или "присваивается", операцию == как "равно" и операцию === как "строго равно".

Операция == является унаследованной возможностью JavaScript и она обоснованно считается источником ошибок. Вы должны почти всегда использовать === вместо == и !== вместо !=.

Как упоминалось в разделе 3.8, объекты JavaScript сравниваются по ссылке, а не по значению. Объект равен самому себе, но не любому другому объекту. Даже если два отдельных объекта имеют одно и то же количество свойств с теми же самыми именами и значениями, то все равно они не равны. Аналогично два массива, которые содержат те же самые элементы в том же порядке, не равны друг другу.

Строгое равенство

Операция строгого равенства === вычисляет свои операнды и сравнивает два полученных значения, как описано ниже, не выполняя никаких преобразований типов.

- Если два значения относятся к разным типам, тогда они не равны.
- Если оба значения являются null или undefined, тогда они равны.
- Если оба значения представляют собой булевское значение true или false, тогда они равны.
- Если одно или оба значения оказались NaN, то они не равны. (Удивительно, но значение NaN никогда не равно никакому другому значению, включая себя самого! Чтобы проверить, имеет ли переменная x значение NaN, применяйте x !== x или глобальную функцию isNaN().)
- Если оба значения являются числами с одной и той же величиной, тогда они равны. Если одно значение представляет собой 0, а другое -0, то они также равны.

- Если оба значения являются строками и содержат в точности те же самые 16-битные величины (см. врезку в разделе 3.3) в тех же позициях, тогда они равны. Если строки отличаются длиной или содержимым, то они не равны. Две строки могут иметь один и тот же смысл и одинаковый внешний вид, но все-таки быть закодированными с использованием разных последовательностей 16-битных значений. Интерпретатор JavaScript не выполняет нормализацию Unicode и для операции === или == строки такого рода не считаются равными.
- Если оба значения ссылаются на тот же самый объект, массив или функцию, тогда они равны. Если два значения ссылаются на разные объекты, они не равны, даже если оба объекта имеют идентичные свойства.

Равенство с преобразованием типов

Операция равенства == похожа на операцию строгого равенства, но менее требовательна. Если значения двух операндов не относятся к одному и тому же типу, тогда она опробует некоторые преобразования типов и пытается выполнить сравнение снова.

- Если два значения имеют один и тот же тип, то они проверяются на предмет строгого равенства, как было описано выше. Если значения строго равны, тогда они равны и нестрого. Если значения не равны строго, то они не равны и нестрого.
- Если два значения не относятся к одному и тому же типу, тогда операция == все еще может счесть их равными. Для проверки на предмет равенства она применяет следующие правила и преобразования типов.
- Если одним значением является `null`, а другим — `undefined`, то они равны.
- Если одно значение представляет собой число, а другое — строку, тогда строка преобразуется в число и предпринимается повторная попытка сравнения с использованием преобразованного значения.
- Если любое из двух значений является `true`, то оно преобразуется в 1 и предпринимается повторная попытка сравнения. Если любое из двух значений является `false`, то оно преобразуется в 0 и предпринимается повторная попытка сравнения.
- Если одним значением оказывается объект, а другим — число или строка, тогда объект преобразуется в элементарное значение с применением алгоритма, описанного в подразделе 3.9.3, и предпринимается повторная попытка сравнения. Объект преобразуется в элементарное значение с помощью либо своего метода `toString()`, либо своего метода `valueOf()`. Встроенные классы базового языка JavaScript опробуют преобразование `valueOf()` перед преобразованием `toString()` за исключением класса `Date`, который выполняет преобразование `toString()`.
- Любые другие комбинации значений не дают равенство.

В качестве примера проверки равенства рассмотрим такое сравнение:

```
"1" == true  // => true
```

Приведенное выражение вычисляется как `true`, указывая на то, что эти два совершенно по-разному выглядящие значения на самом деле равны. Сначала булевское значение `true` преобразуется в число 1 и сравнение выполняется еще раз. Далее строка `"1"` преобразуется в число 1. Поскольку оба значения теперь одинаковы, операция сравнения возвращает `true`.

4.9.2. Операции сравнения

Операции сравнения проверяют относительный порядок (числовой или алфавитный) двух своих операндов.

***Меньше* (<)**

Операция < вычисляется как `true`, если первый операнд меньше второго операнда; иначе она вычисляется как `false`.

***Больше* (>)**

Операция > вычисляется как `true`, если первый операнд больше второго операнда; иначе она вычисляется как `false`.

***Меньше или равно* (<=)**

Операция <= вычисляется как `true`, если первый операнд меньше второго операнда или равен ему; иначе она вычисляется как `false`.

***Больше или равно* (>=)**

Операция >= вычисляется как `true`, если первый операнд больше второго операнда или равен ему; иначе она вычисляется как `false`.

Операнды в операциях сравнения могут быть любого типа. Однако сравнение может выполняться только с числами и строками, поэтому операнды, не являющиеся числами или строками, преобразуются.

Вот как происходят сравнение и преобразование.

- Если любой из двух операндов оказывается объектом, то он преобразуется в элементарное значение, как описано в конце подраздела 3.9.3; если его метод `valueOf()` возвращает элементарное значение, тогда оно и используется. В противном случае применяется возвращаемое значение метода `toString()`.
- Если после любых требуемых преобразований объектов в элементарные значения оба операнда становятся строками, тогда две строки сравниваются в алфавитном порядке, где "алфавитный порядок" определяется числовым порядком 16-битных значений Unicode, которые составляют строки.
- Если после преобразования объекта в элементарное значение, по крайней мере, один операнд оказывается не строковым, тогда оба операнда преобразуются в числа и сравниваются численно. `0` и `-0` считаются равными.

`Infinity` больше, чем любое число за исключением себя, а `-Infinity` меньше, чем любое число за исключением себя. Если один из двух операндов является значением `NaN` (или преобразуется в него), то операция сравнения всегда возвращает `false`. Несмотря на то что арифметические операции не разрешают смешивать значения `BigInt` с обыкновенными числами, операции сравнения допускают сравнения между числами и значениями `BigInt`.

Вспомните, что строки JavaScript представляют собой последовательности 16-битных целочисленных значений, а сравнение строк — это всего лишь числовое сравнение значений в двух строках. Числовой порядок кодирования, определяемый Unicode, может не совпадать с традиционным порядком сопоставления, используемым в любом отдельно взятом языке или локали. В частности, обратите внимание, что сравнение строк чувствительно к регистру, и все буквы ASCII верхнего регистра "больше" всех букв ASCII нижнего регистра. Данное правило может приводить к результатам, которые сбивают с толку, если их не ожидать. Например, в соответствии с операцией < строка "Zoo" (зоопарк) находится перед строкой "aardvark" (африканский муравьед).

Если вам нужен более надежный алгоритм сравнения строк, то испытайте метод `String.localeCompare()`, который также принимает во внимание специфичные к локали определения алфавитного порядка. Для сравнения, нечувствительного к регистру, вы можете преобразовывать все символы в строках к нижнему или верхнему регистру с применением метода `String.toLowerCase()` или `String.toUpperCase()`. А если вас интересует более универсальный и лучше локализованный инструмент сравнения строк, тогда используйте класс `Intl.Collator`, описанный в подразделе 11.7.3.

Как операция +, так и операции сравнения ведут себя по-разному для числовых и строковых операндов. Операция + поддерживает строки: она выполняет конкатенацию, когда любой из двух операндов является строкой. Операции сравнения поддерживают числа и выполняют строковое сравнение, только если оба операнда будут строками:

```
1 + 2       // => 3: сложение
"1" + "2"   // => "12": конкатенация
"1" + 2     // => "12": 2 преобразуется в "2"
11 < 3      // => false: числовое сравнение
"11" < "3"  // => true: строковое сравнение
"11" < 3    // => false: числовое сравнение, "11" преобразуется в 11
"one" < 3   // => false: числовое сравнение, "one" преобразуется в NaN
```

Наконец, имейте в виду, что операции <= (меньше или равно) и >= (больше или равно) при определении, "равны" ли два значения, не полагаются на операции равенства или строгого равенства. Взамен операция <= просто определяется как "не больше", а операция >= — как "не меньше". Единственное исключение возникает, когда любой из операндов имеет значение `NaN` (или преобразуется в него); в таком случае все четыре операции сравнения возвращают `false`.

4.9.3. Операция in

Операция in ожидает с левой стороны операнд, который является строкой, символом или значением, допускающим преобразование в строку. С правой стороны она ожидает операнд, который должен быть объектом. Операция in вычисляется как true, если значением с левой стороны будет имя свойства объекта с правой стороны. Например:

```
let point = {x: 1, y: 1};  // Определение объекта
"x" in point               // => true: объект имеет свойство по имени "x"
"z" in point               // => false: объект не имеет свойства по имени "z"
"toString" in point        // => true: объект наследует метод toString

let data = [7,8,9];        // Массив с элементами (индексами) 0, 1 и 2
"0" in data                // => true: массив имеет элемент "0"
1 in data                  // => true: числа преобразуются в строки
3 in data                  // => false: массив не имеет элемента 3
```

4.9.4. Операция instanceof

Операция instanceof ожидает с левой стороны операнд, который является объектом, а с правой стороны операнд, идентифицирующий класс объектов. Операция вычисляется как true, если объект с левой стороны будет экземпляром класса с правой стороны, и как false в противном случае. В главе 9 рассказывается о том, что в JavaScript классы объектов определяются функцией конструктора, которая их инициализирует. Таким образом, операнд с правой стороны instanceof должен быть функцией. Ниже приведены примеры:

```
let d = new Date();   //Создание нового объекта с помощью конструктора Date()
d instanceof Date     // => true: объект d был создан с помощью Date()
d instanceof Object   // => true: все объекты являются экземплярами Object
d instanceof Number   // => false: d - не объект Number

let a = [1, 2, 3];    // Создание нового массива посредством
                      // синтаксиса литерала типа массива
a instanceof Array    // => true: объект a - массив
a instanceof Object   // => true: все массивы являются объектами
a instanceof RegExp   // => false: массивы - не регулярные выражения
```

Обратите внимание, что все объекты являются экземплярами Object. Принимая решение о том, будет ли объект экземпляром класса, операция instanceof учитывает "суперклассы". Если операнд с левой стороны instanceof — не объект, тогда instanceof возвращает false. Если операнд с правой стороны — не класс объектов, то генерируется ошибка TypeError.

Для понимания работы операции instanceof вы должны знать о "цепочке прототипов". Она представляет собой механизм наследования JavaScript и описана в подразделе 6.3.2. Чтобы вычислить выражение o instanceof f, интерпретатор JavaScript вычисляет f.prototype и затем ищет это значение в цепочке прототипов o. Если оно находится, тогда o — экземпляр f (или какого-то подкласса f) и операция возвращает true.

Если `f.prototype` не входит в состав значений цепочки прототипов `o`, то `o` — не экземпляр `f` и `instanceof` возвращает `false`.

4.10. Логические выражения

Логические операции `&&`, `||` и `!` выполняют действия булевой алгебры и часто применяются в сочетании с операциями отношений для объединения двух выражений отношений в одно более сложное выражение. Логические операции обсуждаются в следующих далее подразделах. Для их полного понимания у вас может возникнуть желание пересмотреть концепции "истинных" и "ложных" значений, введенные в разделе 3.4.

4.10.1. Логическое И (&&)

Операцию `&&` можно осмысливать на трех разных уровнях. На простейшем уровне, когда операция `&&` используется с булевскими операндами, она выполняет булевскую операцию И над двумя значениями и возвращает `true`, если и только если первый *и* второй операнды равны `true`. Если один или оба операнда равны `false`, тогда она возвращает `false`.

Операция `&&` часто применяется в качестве конъюнкции для объединения двух выражений отношений:

```
x === 0 && y === 0  // true, если и только если x и y равны 0
```

Выражения отношений всегда вычисляются как `true` или `false`, поэтому при таком использовании операция `&&` сама возвращает `true` или `false`. Операции отношений имеют более высокий приоритет, чем `&&` (и `||`), так что выражения подобного рода можно безопасно записывать без круглых скобок.

Но операция `&&` не требует, чтобы ее операнды были булевскими значениями. Вспомните, что все значения JavaScript являются либо "истинными", либо "ложными". (За подробностями обращайтесь в раздел 3.4. Ложные значения — это `false`, `null`, `undefined`, `0`, `-0`, `NaN` и `""`. Все остальные значения, включая все объекты, считаются истинными.) Второй уровень осмысления операции `&&` — она представляет собой булевскую операцию И для истинных и ложных значений. Если оба операнда истинные, тогда операция возвращает истинное значение. Иначе один или оба операнда обязаны быть ложными, и операция возвращает ложное значение. В JavaScript любое выражение или оператор, который ожидает булевское значение, будет работать с истинным или ложным значением, поэтому тот факт, что `&&` не всегда возвращает `true` или `false`, не должен вызывать практических проблем.

Обратите внимание, что в приведенном выше описании говорится о том, что операция возвращает "истинное значение" или "ложное значение", но не указано, чем является данное значение. Для этого нам необходимо описать операцию `&&` на третьем, финальном, уровне. Операция `&&` начинается с вычисления своего первого операнда, т.е. выражения с левой стороны. Если оно ложное, то значение целого выражения тоже должно быть ложным, так что `&&` просто возвращает значение слева и даже не вычисляет выражение справа.

С другой стороны, если значение слева истинное, тогда общее значение выражения зависит от значения справа. Если значение справа истинное, то общее значение должно быть истинным, а если значение справа ложное, тогда общее значение должно быть ложным. Таким образом, когда значение слева истинное, операция && вычисляется и возвращает значение справа:

```
let o = {x: 1};
let p = null;
o && o.x  // => 1: o является истинным, так что возвратить значение o.x
p && p.x  // => null: p является ложным, так что возвратить его
          //           и не вычислять p.x
```

Важно понимать, что && может как вычислять, так и не вычислять свой операнд с правой стороны. В приведенном выше примере кода переменная p установлена в null, и выражение p.x в случае вычисления вызвало бы ошибку TypeError. Но код использует && идиоматическим способом, так что p.x вычисляется, только если переменная p истинная — не null или undefined.

Поведение операции && иногда называют коротким замыканием, и временами вы можете видеть код, в котором такое поведение задействовано намеренно для условного выполнения кода. Скажем, следующие две строки кода JavaScript дают эквивалентный эффект:

```
if (a === b) stop();    // Вызвать stop(), только если a === b
(a === b) && stop();    // Здесь делается то же самое
```

В целом вы должны проявлять осторожность всякий раз, когда записываете выражение с побочными эффектами (присваивания, инкрементирования, декрементирования или вызовы функций) с правой стороны операции &&. Произойдут ли эти побочные эффекты, зависит от значения с левой стороны.

Несмотря на довольно сложные аспекты работы операции &&, наиболее часто она применяется как простое действие булевой алгебры, которое имеет дело с истинными и ложными значениями.

4.10.2. Логическое ИЛИ (||)

Операция || выполняет булевскую операцию ИЛИ над своими двумя операндами. Если один или оба операнда истинные, тогда операция возвращает истинное значение. Если оба операнда ложные, то она возвращает ложное значение.

Хотя операция || чаще всего используется просто как булевская операция ИЛИ, подобно операции && она обладает более сложным поведением. Она начинает с вычисления своего первого операнда, т.е. выражения слева. Если значение первого операнда истинное, тогда происходит короткое замыкание и это истинное значение возвращается даже без вычисления выражения справа. С другой стороны, если значение первого операнда ложное, то операция || вычисляет второй операнд и возвращает значение этого выражения.

Как и в случае с операцией &&, вы должны избегать применения с правой стороны операндов, которые включают побочные эффекты, если только вы на-

меренно не хотите воспользоваться тем фактом, что выражение справа может не вычисляться.

Идиоматическое применение операции && связано с выбором первого истинного значения в наборе вариантов:

```
// Если значение maxWidth истинное, то использовать его.
// Иначе искать значение в объекте
// preferences. Если оно не истинное, тогда использовать
// жестко закодированную константу.
let max = maxWidth || preferences.maxWidth || 500;
```

Важно отметить, что если 0 является законным значением для maxWidth, тогда код не будет работать корректно, т.к. 0 — ложное значение. В качестве альтернативы взгляните на операцию ?? (см. подраздел 4.13.2).

До версии ES6 такая идиома часто использовалась в функциях для предоставления параметрам значений по умолчанию:

```
// Копирует свойства o в p и возвращает p
function copy(o, p) {
   p = p || {};     // Если для p не был передан объект,
                    // то использовать вновь созданный объект.
   // Далее следует тело функции.
}
```

Тем не менее, в ES6 и последующих версиях необходимость в этом трюке отпала, поскольку стандартное значение параметра можно просто записывать в самом определении функции: `function copy(o, p={}) { ... }`.

4.10.3. Логическое НЕ (!)

Операция ! является унарной и помещается перед одиночным операндом. Ее цель — инвертирование булевского значения операнда. Скажем, если значение x истинное, то !x вычисляется как false. Если значение x ложное, тогда !x равно true.

В отличие от && и || операция ! преобразует свой операнд в булевское значение (в соответствии с правилами, описанными в главе 3) и затем инвертирует преобразованное значение. Таким образом, операция ! всегда возвращает true или false и вы можете преобразовать любое значение x в эквивалентное булевское значение, применив эту операцию дважды: !!x (см. подраздел 3.9.2).

Будучи унарной, операция ! обладает высоким приоритетом и тесной привязкой. Если вы хотите инвертировать значение выражения наподобие p && q, то должны использовать круглые скобки: !(p && q). Нелишне здесь отметить два закона булевой алгебры, которые мы можем выразить посредством синтаксиса JavaScript:

```
// Законы де Моргана
!(p && q) === (!p || !q)  // => true: для всех значений p и q
!(p || q) === (!p && !q)  // => true: для всех значений p и q
```

4.11. Выражения присваивания

Операция = в JavaScript применяется для присваивания значения переменной или свойству, например:

```
i = 0;          // Установить переменную i в 0
o.x = 1;        // Установить свойство x объекта o в 1
```

Операция = ожидает, что ее операнд с левой стороны будет левым значением: переменной или свойством объекта (или элементом массива). С правой стороны она ожидает, что операнд будет произвольным значением любого типа. Значением выражения присваивания является значение операнда с правой стороны. В качестве побочного эффекта операция = присваивает значение справа переменной или свойству слева, так что будущие ссылки на переменную или свойство вычисляются в это значение.

Хотя выражения присваивания обычно довольно просты, иногда вы можете видеть, что значение выражения присваивания используется в более крупном выражении. Например, вот как можно присвоить и проверить значение в том же самом выражении:

```
(a = b) === 0
```

Поступая так, убедитесь в том, что четко понимаете разницу между операциями = и ===! Обратите внимание, что операция = обладает очень низким приоритетом и в случае, когда ее значение применяется как часть более крупного выражения, обычно нужны круглые скобки.

Операция присваивания имеет ассоциативность справа налево, т.е. при наличии в выражении множества операция присваивания они вычисляются справа налево. Таким образом, с помощью следующего кода можно присвоить одиночное значение множеству переменных:

```
i = j = k = 0;  // Инициализация трех переменных значением 0
```

4.11.1. Присваивание с действием

Помимо нормальной операции присваивания = в JavaScript поддерживается несколько других операций присваивания, которые обеспечивают сокращения за счет комбинирования присваивания с рядом других действий. Скажем, операция += выполняет сложение и присваивание. Следующее выражение:

```
total += salesTax;
```

эквивалентно такому выражению:

```
total = total + salesTax;
```

Как и можно было ожидать, операция += работает с числами или строками. Для числовых операндов она выполняет сложение и присваивание, а для строковых — конкатенацию и присваивание.

Похожие операции включают -=, *=, &= и т.д. Все они перечислены в табл. 4.2.

Таблица 4.2. Операции присваивания

Операция	Пример	Эквивалент
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`
`**=`	`a **= b`	`a = a ** b`
`<<=`	`a <<= b`	`a = a << b`
`>>=`	`a >>= b`	`a = a >> b`
`>>>=`	`a >>>= b`	`a = a >>> b`
`&=`	`a &= b`	`a = a & b`
`|=`	`a |= b`	`a = a | b`
`^=`	`a ^= b`	`a = a ^ b`

В большинстве случаев выражение:

```
a op= b
```

где op — операция, эквивалентно выражению:

```
a = a op b
```

В первой строке выражение a вычисляется один раз, а во второй строке — два раза. Два случая будут отличаться, только если a имеет побочные эффекты, такие как вызов функции или инкрементирование. Например, следующие два присваивания не одинаковы:

```
data[i++] *= 2;
data[i++] = data[i++] * 2;
```

4.12. Вычисление выражений

Как и многие интерпретируемые языки JavaScript обладает возможностью интерпретации строк с исходным кодом JavaScript, вычисляя их с целью выработки значения. JavaScript делает это с помощью глобальной функции `eval()`:

```
eval("3+2")    // => 5
```

Динамическое вычисление строк с исходным кодом — мощное языковое средство, которое почти никогда не требуется на практике. Если вы обнаружили, что используете `eval()`, тогда хорошо подумайте, действительно ли это необходимо. В частности, функция `eval()` может оказаться брешью в безопасности и вы никогда не должны передавать ей строку, полученную из пользовательского ввода. С помощью такого сложного языка, как JavaScript, не существует способа сделать пользовательский ввод безопасным для применения с функцией `eval()`. Из-за проблем с безопасностью некоторые веб-серверы ис-

пользуют HTTP-заголовок Content-Security-Policy, чтобы запретить применение eval() на всем веб-сайте.

В последующих подразделах объясняются основы использования функции eval(), а также две ее ограниченные версии, которые оказывают меньшее влияние на оптимизатор.

eval() — функция или операция?

eval() является функцией, но рассматривается в этой главе, посвященной операциям, т.к. в действительности она должна была быть операцией. Функция eval() была определена в самых ранних версиях языка и с тех пор проектировщики языка и разработчики интерпретатора накладывали на нее ограничения, делая ее все больше и больше похожей на операцию. Современные интерпретаторы JavaScript выполняют много работы по анализу и оптимизации кода. Вообще говоря, если какая-то функция вызывает eval(), то интерпретатор не может ее оптимизировать. Проблема с определением eval() как функции в том, что ей можно давать другие имена:

```
let f = eval;
let g = f;
```

Если это разрешено, то интерпретатор не может точно знать, какие функции вызывают eval(), и энергичная оптимизация невозможна. Проблемы можно было бы избежать, если бы eval() являлась операцией (и зарезервированным словом). В подразделах 4.12.2 и 4.12.3 мы рассмотрим ограничения, налагаемые на функцию eval(), которые делают ее похожей на операцию.

4.12.1. eval()

Функция eval() ожидает один аргумент. Если вы передадите ей значение, отличающееся от строки, то она просто возвратит его. Если вы передадите строку, тогда она попытается выполнить синтаксический разбор строки как кода JavaScript, генерируя ошибку SyntaxError в случае неудачи. Если разбор прошел успешно, то eval() выполнит код и возвратит значение последнего выражения или оператора в строке либо undefined, когда последнее выражение или оператор не имеет значения. Если выполняемая строка генерирует исключение, то оно распространится от вызова до eval().

Ключевой момент, связанный с eval() (при вызове подобным образом), заключается в том, что она использует таблицу переменных кода, который ее вызывает. То есть она ищет значения переменных и определяет новые переменные и функции таким же способом, как поступает локальный код. Если функция определяет локальную переменную x и затем вызывает eval("x"), то будет получено значение этой локальной переменной. Вызов eval("x=1") изменяет значение локальной переменной, а вызов eval("var y = 3;") объявляет новую локальную переменную y. С другой стороны, если в вычисляемой строке применяется let или const, то объявляемая переменная или константа окажется локальной по отношению к контексту eval() и не будет объявлена в вызывающей среде.

Аналогично функция может объявлять локальную функцию посредством такого кода:

```
eval("function f() { return x+1; }");
```

Разумеется, если вызвать функцию `eval()` в коде верхнего уровня, тогда она будет работать с глобальными переменными и глобальными функциями.

Обратите внимание, что строка кода, передаваемая `eval()`, обязана быть синтаксически осмысленной сама по себе: вы не можете использовать `eval()` для вставки фрагментов кода внутрь какой-то функции. Скажем, не имеет смысла записывать `eval("return;")`, поскольку оператор `return` допускается только внутри функций, и тот факт, что вычисляемая строка работает с той же таблицей переменных, что и вызывающая функция, не делает ее частью этой функции. Если ваша строка имеет смысл как автономный сценарий (даже очень короткий вроде `x=0`), то передавать ее `eval()` законно. В противном случае `eval()` сгенерирует ошибку `SyntaxError`.

4.12.2. eval() в глобальном контексте

Способность функции `eval()` изменять локальные переменные создает значительные проблемы оптимизаторам JavaScript. Однако в качестве обходного приема интерпретаторы просто делают меньший объем оптимизации для любой функции, которая вызывает `eval()`. Но что должен делать интерпретатор JavaScript, если сценарий определяет псевдоним для функции `eval()` и затем вызывает ее под другим именем? В спецификации JavaScript заявлено, что когда функция `eval()` вызывается под именем, отличающимся от "eval", то она должна вычислить строку, как если бы все происходило в глобальном коде. Вычисляемый код может определять новые глобальные переменные или глобальные функции, а также устанавливать глобальные переменные, но он не будет использовать или модифицировать любые переменные, локальные по отношению к вызывающей функции, и потому не будет препятствовать оптимизации.

"Прямой вызов" — это вызов функции `eval()` с помощью выражения, применяющего точное, безусловное имя "eval" (которое начинает восприниматься как зарезервированное слово). Прямые вызовы `eval()` используют таблицу переменных вызывающего контекста. Любой другой вызов — непрямой вызов — применяет в качестве таблицы переменных глобальный объект и не может читать, устанавливать или определять локальные переменные или функции. (И прямой, и непрямой вызовы могут определять новые переменные только посредством `var`. Использование `let` и `const` внутри вычисляемой строки приводит к созданию переменных и констант, которые являются локальными по отношению к вызову `eval()` и не изменяют вызывающую или глобальную среду.)

Сказанное демонстрируется в следующем коде:

```
const geval = eval;                 // Использование другого имени делает
                                    // eval в глобальном контексте
let x = "global", y = "global";     // Две глобальные переменные

function f() {                      // Эта функция делает eval в локальном
                                    // контексте
   let x = "local";                 // Определение локальной переменной
   eval("x += 'changed';");         // Прямой вызов eval устанавливает
                                    // локальную переменную
   return x;                        // Возвращение измененной локальной
                                    // переменной
}
function g() {      // Эта функция делает eval в глобальном контексте
   let y = "local";                 // Локальная переменная
   geval("y += 'changed';");        // Непрямой вызов eval устанавливает
                                    // глобальную переменную
   return y;                        // Возвращение неизмененной локальной
                                    // переменной
}
console.log(f(), x);                // Изменение локальной переменной:
                                    // выводится "localchanged global":
console.log(g(), y);                // Изменение глобальной переменной:
                                    // выводится "local globalchanged":
```

Обратите внимание, что возможность вызывать `eval()` в глобальном контексте — не просто приспосабливание под нужды оптимизатора; в действительности это чрезвычайно полезное средство, которое позволяет выполнять строки кода, как если бы они были независимыми сценариями верхнего уровня. Как отмечалось в начале текущего раздела, вычисление строки кода редко требуется по-настоящему. Но если вам оно понадобится, тогда более вероятно, что вы захотите делать вызовы `eval()` в глобальном контексте, а не локальном.

4.12.3. `eval()` в строгом режиме

Строгий режим (см. подраздел 5.6.3) налагает добавочные ограничения на поведение функции `eval()` и даже на применение идентификатора "eval". Когда `eval()` вызывается в коде строгого режима или вычисляемая строка кода сама начинается с директивы `use strict`, производится вызов `eval()` в локальном контексте с закрытой таблицей переменных. Это означает, что в строгом режиме вычисляемая строка кода может запрашивать и устанавливать локальные переменные, но не может определять новые переменные или функции в локальном контексте.

Кроме того, строгий режим делает `eval()` еще более похожей на операцию, фактически превращая "eval" в зарезервированное слово. Вам не разрешено замещать функцию `eval()` новым значением. И вам не разрешено объявлять переменную, функцию, параметр функции или параметр блока `catch` с именем "eval".

4.13. Смешанные операции

JavaScript поддерживает несколько смешанных операций, описанных в последующих подразделах.

4.13.1. Условная операция (? :)

Условная операция — единственная операция с тремя операндами в JavaScript, которая иногда называется *тернарной операцией*. Временами ее записывают как ? :, хотя в коде она выглядит не совсем так. Поскольку эта операция имеет три операнда, первый находится перед ?, второй — между ? и :, а третий — после :. Вот как она используется:

```
x > 0 ? x : -x    // Абсолютное значение x
```

Операнды условной операции могут быть любого типа. Первый операнд вычисляется и интерпретируется как булевское значение. Если значение первого операнда истинное, тогда вычисляется второй операнд и возвращается его значение. В противном случае, когда значение первого операнда ложное, то вычисляется третий операнд и возвращается его значение. Из второго и третьего операндов вычисляется только один, но никогда оба.

Наряду с тем, что вы можете достичь сходных результатов с применением оператора `if` (см. подраздел 5.3.1), операция `?:` зачастую предлагает удобное сокращение. Ниже демонстрируется типичное использование, где проверяется, определена ли переменная (и имеет ли она содержательное, истинное значение), и если да, то применяется ее значение, а иначе стандартное:

```
greeting = "hello " + (username ? username : "there");
```

Код эквивалентен следующему оператору `if`, но более компактен:

```
greeting = "hello ";
if (username) {
   greeting += username;
} else {
   greeting += "there";
}
```

4.13.2. Операция выбора первого определенного операнда (??)

Операция выбора первого определенного операнда `??` вычисляется как ее первый определенный операнд: если левый операнд не является `null` или `undefined`, тогда она возвращает его значение. Иначе операция `??` возвращает значение правого операнда. Подобно `&&` и `||` операция?? работает по принципу короткого замыкания: она вычисляет второй операнд, только если первый операнд оказался `null` или `undefined`. Если выражение a не имеет побочных эффектов, тогда выражение a `??` b эквивалентно:

```
(a !== null && a !== undefined) ? a : b
```

Операция ?? является удобной альтернативой операции || (см. подраздел 4.10.2), когда вы хотите выбрать первый *определенный*, а не первый истинный операнд. Хотя || — формально операция логического ИЛИ, она также используется идиоматически для выбора первого неложного операнда с помощью такого кода:

```
// Если значение maxWidth истинное, то использовать его.
// Иначе искать значение в объекте preferences. Если оно не истинное,
// тогда использовать жестко закодированную константу.
let max = maxWidth || preferences.maxWidth || 500;
```

Проблема с таким идиоматическим применением состоит в том, что ноль, пустая строка и false — ложные значения, которые в определенных обстоятельствах могут быть совершенно допустимыми. Если в приведенном выше примере кода maxWidth равно нулю, то это значение будет проигнорировано. Но если мы изменим операцию || на ??, тогда получим выражение, где ноль оказывается допустимым значением:

```
// Если значение maxWidth определено, то использовать его.
// Иначе искать значение в объекте preferences. Если оно
// не определено, тогда использовать жестко закодированную константу.
let max = maxWidth ?? preferences.maxWidth ?? 500;
```

Далее приведены дополнительные примеры, которые показывают, как работает операция ??, когда первый операнд ложный. Если первый операнд ложный, но определен, тогда операция ?? возвращает его. Только когда первый операнд является "нулевым" (т.е. null или undefined), операция ?? вычисляет и возвращает второй операнд:

```
let options = { timeout: 0, title: "", verbose: false, n: null };
options.timeout ?? 1000     // => 0: как определено в объекте
options.title ?? "Untitled" // => "": как определено в объекте
options.verbose ?? true     // => false: как определено в объекте
options.quiet ?? false      // => false: свойство не определено
options.n ?? 10             // => 10: свойство равно null
```

Обратите внимание, что выражения timeout, title и verbose имели бы другие значения, если бы вместо ?? использовалась операция ||.

Операция ?? похожа на операции && и ||, но ее приоритет не выше и не ниже, чем у них. Если вы применяете операцию ?? в выражении с одной из этих операций, тогда должны использовать явные круглые скобки для указания, какое действие нужно выполнить первым:

```
(a ?? b) || c               // Сначала ??, затем ||
a ?? (b || c)               // Сначала ||, затем ??
a ?? b || c                 // SyntaxError: круглые скобки обязательны
```

Операция ?? определена в ES2020 и по состоянию на начало 2020 поддерживалась в текущих или бета-версиях большинства ведущих браузеров. Формально она называется операцией "объединения с null", но я избегаю такого термина, потому что эта операция выбирает один из своих операндов, но не "объединяет" их каким-то заметным для меня образом.

4.13.3. Операция typeof

typeof — унарная операция, помещаемая перед своим единственным операндом, который может быть любого типа. Ее значением является строка, которая указывает тип операнда. А табл. 4.3 приведены значения операции typeof для всех значений JavaScript.

Таблица 4.3. Значения, возвращаемые операцией typeof

x	typeof x
undefined	"undefined"
null	"object"
true или false	"boolean"
Любое число или NaN	"number"
Любое значение BigInt	"bigint"
Любая строка	"string"
Любой символ	"symbol"
Любая функция	"function"
Любой объект, отличный от функции	"object"

Вы можете применять операцию typeof в выражении такого рода:

```
// Если значением является строка, тогда поместить ее в кавычки,
// иначе преобразовать
(typeof value === "string") ? "'" + value + "'" : value.toString()
```

Обратите внимание, что typeof возвращает "object", когда операнд имеет значение null. Если вы хотите проводить различие между null и объектами, тогда должны предусмотреть явную проверку значения для такого особого случая.

Хотя функции JavaScript являются разновидностью объектов, операция typeof считает их достаточно отличающимися, чтобы обеспечить им собственное возвращаемое значение.

Поскольку операция typeof выдает "object" для всех объектов и массивов кроме функций, она позволяет лишь проводить различие между объектами и элементарными типами. Чтобы различать один класс объектов от другого, вам придется использовать другие методики, такие как операция instanceof (см. подраздел 4.9.4), атрибут class (см. подраздел 14.4.3) или свойство constructor (см. подраздел 9.2.2 и раздел 14.3).

4.13.4. Операция delete

delete — унарная операция, которая пытается удалить свойство объекта или элемент массива, указанный в качестве ее операнда. Подобно операциям присваивания, инкремента и декремента операция delete обычно применяется ради ее побочного эффекта, связанного с удалением свойства, а не возвращаемого значения.

Вот несколько примеров:

```
let o = { x: 1, y: 2};  // Начать с какого-то объекта
delete o.x;             // Удалить одно из его свойств
"x" in o                // => false: свойство больше не существует

let a = [1,2,3];        // Начать с какого-то массива
delete a[2];            // Удалить последний элемент массива
2 in a     // => false: элемент 2 массива больше не существует
a.length   // => 3: обратите внимание, что длина массива не изменилась
```

Важно отметить, что удаленное свойство или элемент массива не просто устанавливается в значение undefined. Когда свойство удаляется, оно перестает существовать. При попытке чтения несуществующего свойства возвращается undefined, но вы можете проверить, существует ли свойство, с помощью операции in (см. подраздел 4.9.3). Удаление элемента массива оставляет в массиве "дыру" и не изменяет длину массива. Результирующий массив становится *разреженным* (см. раздел 7.3).

Операция delete ожидает, что операнд будет левым значением. Если это не так, тогда она не предпринимает никаких действий и возвращает true. В противном случае delete пытается удалить указанное левое значение. Операция delete возвращает true, если указанное левое значение успешно удалено. Тем не менее, удалять можно не все свойства: неконфигурируемые свойства (см. раздел 14.1) защищены от удаления.

В строгом режиме операция delete генерирует ошибку SyntaxError, если ее операнд является неуточненным идентификатором, таким как переменная, функция или параметр функции: она работает, только когда операнд представляет собой выражение доступа к свойству (см. раздел 4.4). Строгий режим также предписывает, что delete генерирует TypeError, если запрашивается удаление любого неконфигурируемого (т.е. неудаляемого) свойства. За рамками строгого режима исключения в таких случаях не возникают, и delete просто возвращает false, указывая на то, что операнд не может быть удален.

Ниже приведено несколько примеров использования операции delete:

```
let o = {x: 1, y: 2};
delete o.x;    // Удаление одного из свойств объекта; возвращается true
typeof o.x;    // Свойство не существует; возвращается "undefined"
delete o.x;    // Удаление несуществующего свойства; возвращается true
delete 1;      // Это не имеет смысла, но просто возвращается true

// Удалять переменную нельзя; возвращается false
// или генерируется SyntaxError в строгом режиме
delete o;

// Неудаляемое свойство: возвращается false
// или генерируется TypeError в строгом режиме
delete Object.prototype;
```

Мы снова встретимся с операцией delete в разделе 6.4.

4.13.5. Операция await

Операция await появилась в ES2017 как способ сделать асинхронное программирование в JavaScript более естественным. Для ее понимания понадобится читать главу 13. Однако говоря кратко, await ожидает в качестве своего единственного операнда объект Promise (представляющий асинхронное вычисление) и заставляет программу вести себя так, будто она ждет окончания асинхронного вычисления (но делает это без фактического блокирования и не препятствует одновременному выполнению остальных асинхронных операций). Значением операции await является значение завершения объекта Promise. Важно отметить, что await допускается применять только внутри функций, которые были объявлены асинхронными с помощью ключевого слова async. Полные детали ищите в главе 13.

4.13.6. Операция void

void — унарная операция, располагающаяся перед своим операндом, который может быть любого типа. Операция void необычна и используется редко; она вычисляет свой операнд, после чего отбрасывает значение и возвращает undefined. Поскольку значение операнда отбрасывается, применение void имеет смысл, только когда у операнда есть побочные эффекты.

Операция void настолько неясна, что придумать практический пример ее использования нелегко. Один из сценариев мог бы предусматривать определение функции, которая ничего не возвращает, но также применяет сокращенный синтаксис стрелочных функций (см. подраздел 8.1.3), где тело функции представляет собой единственное выражение, подлежащее вычислению и возвращению. Если вы вычисляете выражение исключительно ради его побочных эффектов и не хотите возвращать его значение, тогда самое простое, что вы можете сделать — использовать фигурные скобки вокруг тела функции. Но в качестве альтернативы в данном случае вы могли бы применить операцию void:

```
let counter = 0;
const increment = () => void counter++;
increment()                                    // => undefined
counter                                        // => 1
```

4.13.7. Операция "запятая"

Операция "запятая" (,) является бинарной операцией, чьи операнды могут быть любого типа. Она вычисляет свой левый операнд, затем правый операнд и далее возвращает значение правого операнда. Таким образом, следующая строка кода:

```
i=0, j=1, k=2;
```

вычисляется как 2 и по существу эквивалентна:

```
i = 0; j = 1; k = 2;
```

Выражение слева всегда вычисляется, но его значение отбрасывается, т.е. использование операции "запятая" имеет смысл, только если у выражения слева имеются побочные эффекты. Единственная ситуация, в которой обычно применяется операция "запятая", связана с циклом `for` (см. подраздел 5.4.3), содержащим множество переменных цикла:

```
// Первая запятая является частью синтаксиса оператора let.
// Вторая запятая - это операция "запятая": она позволяет поместить
// два выражения (i++ и j--) в один оператор (цикл for), который
// ожидает одно выражение.
for(let i=0,j=10; i < j; i++,j--) {
   console.log(i+j);
}
```

4.14. Резюме

В главе был охвачен широкий спектр тем и предложено много справочного материала, который вы можете захотеть перечитать в будущем, продолжая изучать JavaScript. Ниже перечислены основные моменты, рассмотренные в главе, которые следует запомнить.

- Выражения — это синтаксические конструкции (или фразы) программы JavaScript.
- Любое выражение может быть вычислено как значение JavaScript.
- Вдобавок к выпуску значений выражения также могут иметь побочные эффекты (такие как присваивание переменных).
- Простые выражения вроде литералов, ссылок на переменные и доступа к свойствам могут комбинироваться с помощью операций, давая более крупные выражения.
- В JavaScript определены операции для арифметических действий, сравнений, булевой логики, присваивания и манипулирования битами, а также ряд смешанных операций, включая тернарную условную операцию.
- Операция + используется для сложения чисел и конкатенации строк.
- Логические операции `&&` и `||` обладают особым поведением "короткого замыкания" и временами вычисляют только один из своих аргументов. Общепринятые идиомы JavaScript требуют от вас понимания особого поведения этих операций.

ГЛАВА 5

Операторы

В главе 4 выражения описывались как синтаксические конструкции или фразы JavaScript. Продолжая такую аналогию, *операторы* являются предложениями или командами JavaScript. Подобно тому, как предложения английского языка завершаются и отделяются друг от друга точками, предложения JavaScript заканчиваются точками с запятой (см. раздел 2.6). Выражения *вычисляются* для выдачи значения, но операторы *выполняются*, чтобы что-то произошло.

Один из способов "заставить, чтобы что-то произошло" предусматривает вычисление выражения, которое имеет побочные эффекты. Выражения с побочными эффектами, такими как присваивание и вызов функции, могут выступать в качестве операторов, и в случае использования подобным образом они называются *операторами-выражениями*. Похожая категория операторов называется *операторами объявлений*, которые объявляют новые объявления и определяют новые функции.

Программа JavaScript представляет собой не более чем последовательность операторов, подлежащих выполнению. По умолчанию интерпретатор JavaScript выполняет эти операторы друг за другом в порядке, в котором они записаны. Другой способ "заставить, чтобы что-то произошло" заключается в изменении стандартного порядка выполнения, для чего в JavaScript есть несколько операторов или управляющих структур, которые описаны ниже.

Условные операторы

Операторы вроде `if` и `switch`, которые вынуждают интерпретатор JavaScript выполнять или пропускать операторы в зависимости от значения выражения.

Операторы циклов

Операторы наподобие `while` и `for`, которые обеспечивают многократное выполнение других операторов.

Операторы переходов

Операторы вроде `break`, `return` и `throw`, которые заставляют интерпретатор переходить к другой части программы.

В последующих разделах описаны разнообразные операторы в JavaScript и объяснен их синтаксис. В табл. 5.1, приведенной в конце главы, предлагается сводка по синтаксису. Программа JavaScript является просто последовательностью операторов, отделенных друг от друга точками с запятой, так что после освоения операторов JavaScript вы можете приступать к написанию программ на языке JavaScript.

5.1. Операторы-выражения

Простейший вид операторов в JavaScript — выражения, которые имеют побочные эффекты. Такая разновидность операторов демонстрировалась в главе 4. Одной из основных категорий операторов-выражений считаются операторы присваивания, например:

```
greeting = "Hello " + name;
i *= 3;
```

Операции инкремента и декремента, ++ и --, относятся к операторам-выражениям. Их побочный эффект заключается в том, что они изменяют значение переменной, как если бы выполнялось присваивание:

```
counter++;
```

Важный побочный эффект операции `delete` связан с удалением свойства объекта. Таким образом, она почти всегда применяется как оператор, а не часть более крупного выражения:

```
delete o.x;
```

Вызовы функций — еще одна основная категория операторов-выражений. Например:

```
console.log(debugMessage);
displaySpinner();  // Гипотетическая функция для отображения
                   // циклического счетчика в веб-приложении
```

Такие вызовы функций являются выражениями, но имеют побочные эффекты, которые воздействуют на размещающую среду или на состояние программы, и здесь они используются как операторы. Если побочные эффекты в функции отсутствуют, тогда нет никакого смысла вызывать ее кроме ситуации, когда вызов представляет собой часть более крупного выражения или оператора присваивания. Скажем, вы не должны вычислять косинус и отбрасывать результат:

```
Math.cos(x);
```

Но вы можете вычислить значение и присвоить его переменной для применения в будущем:

```
cx = Math.cos(x);
```

Обратите внимание, что каждая строка в каждом примере оканчивается точкой с запятой.

5.2. Составные и пустые операторы

Подобно тому, как операция "запятая" (см. подраздел 4.13.7) комбинирует множество выражений в одно выражение, *операторный блок* объединяет множество операторов в единственный *составной оператор*. Операторный блок — это просто последовательность операторов, помещенная внутрь фигурных скобок. Соответственно, следующие строки кода действуют как один оператор и могут использоваться везде, где JavaScript ожидает одиночный оператор:

```
{
    x = Math.PI;
    cx = Math.cos(x);
    console.log("cos(π) = " + cx);
}
```

О приведенном операторном блоке нужно сделать несколько замечаний. Во-первых, он *не* оканчивается точкой с запятой. Элементарные операторы в блоке завершаются точками с запятой, но сам блок — нет. Во-вторых, строки внутри блока записываются с отступом относительно фигурных скобок, в которые он заключен. Поступать так необязательно, но в итоге код становится более простым для чтения и понимания. Подобно тому, как выражения часто содержат подвыражения, многие операторы JavaScript содержат подоператоры. Формально синтаксис JavaScript обычно разрешает одиночный подоператор. Например, синтаксис цикла `while` включает одиночный оператор, который служит телом цикла. С применением операторного блока вы можете поместить внутрь разрешенного одиночного подоператора любое количество операторов.

Составной оператор позволяет использовать множество операторов там, где синтаксис JavaScript ожидает одиночный оператор. Противоположностью будет *пустой оператор*: он дает возможность не включать операторы там, где ожидается один оператор. Вот как выглядит пустой оператор:

```
;
```

При выполнении пустого оператора интерпретатор JavaScript не предпринимает никаких действий. Пустой оператор иногда полезен, когда желательно создать цикл с пустым телом. Взгляните на следующий цикл `for` (циклы `for` будут раскрываться в подразделе 5.4.3):

```
// Инициализировать массив a
for(let i = 0; i < a.length; a[i++] = 0) ;
```

Вся работа в данном цикле делается выражением `a[i++] = 0`, а потому отсутствует необходимость в теле цикла. Однако синтаксис JavaScript требует оператор в качестве тела цикла, поэтому применяется пустой оператор — просто точка с запятой. Обратите внимание, что случайное помещение точки с запятой после правой круглой скобки цикла `for`, цикла `while` или оператора `if` может приводить к неприятным ошибкам, которые трудно выявлять. Например, вряд ли показанный ниже код делает то, что намеревался его автор:

```
if ((a === 0) || (b === 0));      // Ой! Эта строка ничего не делает...
    o = null;                     // а эта строка выполняется всегда
```

Когда вы намеренно используете пустой оператор, хорошей идеей будет снабдить ваш код комментариями, чтобы было ясно, что вы поступаете подобным образом осознанно. Вот пример:

```
for(let i = 0; i < a.length; a[i++] = 0) /* пустое тело */ ;
```

5.3. Условные операторы

Условные операторы выполняют или пропускают другие операторы в зависимости от значения указанного выражения. Такие операторы представляют собой точки принятия решений в коде и временами также известны как "ветвления". Если вы подумаете о прохождении интерпретатора JavaScript через ваш код, то условные операторы являются местами, где код разветвляется по двум и более путям, а интерпретатор обязан выбирать, по какому пути следовать.

В последующих подразделах объясняется базовый условный оператор `if/else`, а также раскрывается `switch` — более сложный оператор разветвления по множеству путей.

5.3.1. `if`

Оператор `if` — это фундаментальный управляющий оператор, который позволяет интерпретатору JavaScript принимать решения или, говоря точнее, выполнять операторы условным образом. Он имеет две формы. Вот первая:

```
if (выражение)
   оператор
```

В такой форме сначала вычисляется *выражение*. Если результирующее значение истинно, тогда *оператор* выполняется. Если *выражение* ложно, то *оператор* не выполняется. (Определение истинных и ложных значений было дано в разделе 3.4.) Например:

```
if (username == null)      // Если имя пользователя null или undefined,
   username = "John Doe"; // тогда определить его
```

Или аналогично:

```
// Если имя пользователя null, undefined, false, 0, "" или NaN,
// то присвоить ему новое значение
if (!username) username = "John Doe";
```

Обратите внимание, что круглые скобки вокруг *выражения* являются обязательной частью синтаксиса оператора `if`.

Синтаксис JavaScript требует наличия после ключевого слова `if` единственного оператора и выражения в круглых скобках, но вы можете применить операторный блок для объединения множества операторов в один. Таким образом, оператор `if` может также выглядеть следующим образом:

```
if (!address) {
   address = "";
   message = "Пожалуйста, укажите почтовый адрес.";
}
```

Во второй форме оператора `if` вводится конструкция `else`, которая выполняется, когда *выражение* равно `false`. Вот ее синтаксис:

```
if (выражение)
   оператор1
else
   оператор2
```

Такая форма оператора `if` выполняет *оператор1*, если *выражение* истинно, и *оператор2*, если оно ложно. Например:

```
if (n === 1)
   console.log("Вы получили 1 новое сообщение.");
else
   console.log(`Вы получили ${n} новых сообщений.`);
```

При наличии вложенных операторов `if` с конструкциями `else` важно гарантировать то, что конструкции `else` относятся к надлежащим операторам `if`. Взгляните на следующие строки кода:

```
i = j = 1;
k = 2;
if (i === j)
   if (j === k)
      console.log("i равно k");
else
   console.log("i не равно j");     // НЕПРАВИЛЬНО!!
```

В приведенном примере внутренний оператор `if` формирует одиночный оператор, разрешенный синтаксисом внешнего оператора `if`. К сожалению, не совсем ясно (кроме как из подсказки, которую дают отступы), какому оператору `if` принадлежит конструкция `else`. И в этом примере отступы сделаны неправильно, потому что интерпретатор JavaScript в действительности трактует предыдущий пример так:

```
if (i === j) {
   if (j === k)
      console.log("i равно k");
   else
      console.log("i не равно j");  // ОЙ!
}
```

Правило в JavaScript (и в большинстве языков программирования) заключается в том, что конструкция `else` является частью ближайшего оператора `if`. Чтобы пример стал менее двусмысленным и более простым для чтения, понимания, сопровождения и отладки, вы должны использовать фигурные скобки:

```
if (i === j) {
   if (j === k) {
      console.log("i равно k");
   }
} else {  // Вот к чему приводит размещение фигурной скобки!
   console.log("i не равно j");
}
```

Многие программисты имеют привычку заключать тела операторов `if` и `else` (а также других составных операторов вроде циклов `while`) в фигурные скобки, даже когда тело состоит из единственного оператора. Такой подход может предотвратить проблему, о которой только что говорилось, и я советую вам принять такую практику. В печатной книге я вынужден обеспечивать компактность кода примеров по вертикали, а потому в этом плане сам не всегда следую собственному совету.

5.3.2. `else if`

Оператор `if/else` вычисляет выражение и в зависимости от результата выполняет одну из двух порций кода. Но что делать, когда нужно выполнить одну из многих порций кода? Один из способов добиться цели предусматривает применение оператора `else if`. По правде говоря, `else if` является не оператором JavaScript, а часто используемой программной идиомой, которая возникает при многократном применении операторов `if/else`:

```
if (n === 1) {
    // Выполнить блок кода #1
} else if (n === 2) {
    // Выполнить блок кода #2
} else if (n === 3) {
    // Выполнить блок кода #3
} else {
    // Если ни одна из конструкций else не выполнилась, тогда выполнить
блок кода #4
}
```

В этом коде нет ничего особенного. Он представляет собой просто последовательность операторов `if`, где каждый следующий `if` является частью конструкции `else` предыдущего оператора. Использовать идиому `else if` предпочтительнее и понятнее, чем записывать такие операторы в синтаксически эквивалентной полностью вложенной форме:

```
if (n === 1) {
    // Выполнить блок кода #1
}
else {
    if (n === 2) {
        // Выполнить блок кода #2
    }
    else {
        if (n === 3) {
            // Выполнить блок кода #3
        }
        else {
            // Если ни одна из конструкций else не выполнилась,
            // тогда выполнить блок кода #4
        }
    }
}
```

5.3.3. switch

Оператор if вызывает ветвление в потоке управления программы, а с помощью идиомы else if вы можете обеспечить ветвление многими путями. Тем не менее, такое решение не будет наилучшим, когда все ветви зависят от значения одного и того же выражения. В этом случае неэкономно многократно вычислять такое выражение во множестве операторов if.

Именно для данной ситуации предназначен оператор switch. За ключевым словом switch следует выражение в круглых скобках и блок кода в фигурных скобках:

```
switch(выражение) {
   операторы
}
```

Однако полный синтаксис оператора switch сложнее показанного здесь. Различные места в блоке кода помечаются ключевым словом case, за которым следует выражение и двоеточие. Когда оператор switch выполняется, он вычисляет значение *выражения* и затем ищет метку case, результат вычисления выражения которой дает то же самое значение (одинаковость определяется операцией ===). В случае ее нахождения он начинает выполнять блок кода с оператора, помеченного case. Если метка case с совпадающим значением не найдена, тогда он ищет оператор, помеченный как default:. При отсутствии метки default: оператор switch вообще пропускает блок кода.

Объяснить работу оператора switch словами довольно сложно; все становится гораздо яснее, если рассмотреть пример. Приведенный далее оператор switch эквивалентен множеству операторов if/else из предыдущего раздела:

```
switch(n) {
case 1:                          // Начать здесь, если n === 1
   // Выполнить блок кода #1
   break;                        // Остановиться здесь
case 2:                          // Начать здесь, если n === 2
   // Выполнить блок кода #2
   break;                        // Остановиться здесь
case 3:                          // Начать здесь, если n === 3
   // Выполнить блок кода #3
   break;                        // Остановиться здесь
default:                         // Если все остальное не выполнилось...
   // Выполнить блок кода #4
   break;                        // Остановиться здесь
}
```

Обратите внимание на ключевое слово break, применяемое в конце каждого case в данном коде. Оператор break, описанный позже в главе, заставляет интерпретатор перейти в конец оператора switch (или "прервать" его) и продолжить выполнение со следующего за ним оператора. Конструкция case в операторе switch указывает только *начальную точку* желаемого кода, но не задают какую-то конечную точку. В отсутствие операторов break оператор switch начинает выполнение блока кода с метки case, значение выражения которой

совпадает со значением выражения самого switch, и продолжает выполнять операторы, пока не достигнет конца блока. В редких ситуациях подобный код, "проваливающийся" с одной метки case в следующую, записывать полезно, но в 99% случаев вы должны аккуратно завершать каждую метку case оператором break. (Тем не менее, когда switch используется внутри функции, вы можете применять оператор return вместо break. Оба они предназначены для прекращения оператора switch и предотвращают проваливание потока выполнения в следующую метку case.)

Ниже показан более реалистичный пример оператора switch; он преобразует значение в строку способом, который зависит от типа значения:

```
function convert(x) {
   switch(typeof x) {
   case "number":  // Преобразовать число в шестнадцатеричное целое
      return x.toString(16);
   case "string":  // Возвратить строку, заключенную в кавычки
      return '"' + x + '"';
   default:        // Преобразовать любой другой тип обычным образом
      return String(x);
   }
}
```

В предшествующих двух примерах за ключевыми словами case следовали соответственно числовые и строковые литералы. Именно так оператор switch чаще всего используется на практике, но имейте в виду, что стандарт ECMAScript разрешает указывать после case произвольное выражение.

Оператор switch сначала вычисляет выражение, следующее за ключевым словом switch, после чего вычисляет выражения case в порядке их появления до тех пор, пока не найдет совпадающее значение[1]. Совпадение определяется с применением операции идентичности ===, а не операции равенства ==, поэтому выражения должны совпадать без какого-либо преобразования типов.

Поскольку не все выражения case вычисляются при каждом выполнении оператора switch, вы должны избегать использования выражений case, которые имеют побочные эффекты, такие как вызовы функций или присваивания. Безопаснее всего просто ограничивать выражения case константными выражениями.

Как объяснялось ранее, если ни одно выражение case не дает совпадения с выражением switch, тогда оператор switch начинает выполнение своего тела с оператора, помеченного как default:. Если метка default: отсутствует, то switch вообще пропускает свое тело. Обратите внимание, что в приведенных примерах метка default: находилась в конце тела switch после всего меток case. Это логичное и обычное место для нее, но в действительности она может появляться где угодно внутри тела оператора.

[1] Тот факт, что выражения case вычисляются во время выполнения, делает оператор switch языка JavaScript сильно отличающимся от оператора switch языков C, C++ и Java (и вдобавок менее эффективным). В C, C++ и Java выражения case обязаны быть константами времени компиляции того же самого типа, а операторы switch часто компилируются в высокоэффективные *таблицы переходов*.

5.4. Циклы

Чтобы понять условные операторы, мы представляли их себе как ответвления пути исходного кода, по которому движется интерпретатор JavaScript. *Операторы цикла* можно представить как развороты обратно с целью повторения порций кода. В JavaScript имеется пять операторов цикла: `while`, `do/while`, `for`, `for/of` (и его разновидность `for/await`) и `for/in`. Все они по очереди объясняются в последующих подразделах. Распространенным сценарием применения циклов является проход по элементам массива. В разделе 7.6 подробно обсуждается цикл такого рода, а также раскрываются специальные методы организации циклов, определенные в классе `Array`.

5.4.1. `while`

В точности как оператор `if` считается базовым условным оператором JavaScript, оператор `while` представляет базовый цикл JavaScript. Он имеет следующий синтаксис:

```
while (выражение)
  оператор
```

Для выполнения оператора `while` интерпретатор сначала вычисляет *выражение*. Если значение выражения ложное, тогда интерпретатор пропускает *оператор*, служащий телом цикла, и переходит к следующему оператору в программе. С другой стороны, если *выражение* истинно, то интерпретатор выполняет *оператор* и организует повторение, переходя в начало цикла и заново вычисляя *выражение*. По-другому можно сказать, что интерпретатор многократно выполняет *оператор*, пока *выражение* истинно. Обратите внимание, что с помощью синтаксиса `while(true)` можно создать бесконечный цикл.

Вряд ли вы захотите, чтобы интерпретатор JavaScript выполнял одну и ту же операцию бесчисленное количество раз. Почти в каждом цикле на каждой *итерации* изменяется одна или несколько переменных. Так как переменные изменяются, действия, предпринимаемые в результате выполнения *оператора*, могут отличаться при каждом проходе цикла. Более того, если изменяемая переменная или переменные присутствуют в *выражении*, то значение выражения может быть разным каждый раз, когда проходится цикл. Это важно, иначе выражение, которое начинается с истинного значения, никогда не изменится, а цикл никогда не закончится! Далее приведен пример цикла `while`, выводящего числа от 0 до 9:

```
let count = 0;
while(count < 10) {
  console.log(count);
  count++;
}
```

Как видите, переменная `count` начинает со значения 0 и инкрементируется при каждом выполнении тела цикла. После десятикратного выполнения тела цикла выражение становится ложным (т.е. переменная `count` перестает быть меньше 10), оператор `while` завершается и интерпретатор может переходить к

следующему оператору в программе. Многие циклы имеют переменную счетчика вроде count. В качестве счетчиков циклов обычно используются имена переменных i, j и k, хотя вы должны выбирать более описательные имена, если они делают ваш код более легким для понимания.

5.4.2. do/while

Цикл do/while похож на цикл while, но только выражение цикла проверяется в конце цикла, а не в начале. Таким образом, тело цикла всегда выполняется, по крайней мере, один раз. Вот его синтаксис:

```
do
   оператор
while (выражение);
```

Цикл do/while применяется не так часто, как while — на практике довольно редко имеется уверенность в том, что цикл желательно выполнить хотя бы раз. Ниже показан пример цикла do/while:

```
function printArray(a) {
   let len = a.length, i = 0;
   if (len === 0) {
      console.log("Пустой массив");
   } else {
      do {
         console.log(a[i]);
      } while(++i < len);
   }
}
```

Между циклом do/while и обыкновенным циклом while есть пара синтаксических различий. Во-первых, цикл do требует наличия как ключевого слова do (для пометки начала цикла), так и ключевого слова while (для пометки конца и ввода условия цикла). Во-вторых, цикл do всегда должен завершаться точкой с запятой. Цикл while не нуждается в точке с запятой, если тело цикла заключено в фигурные скобки.

5.4.3. for

Оператор for предоставляет циклическую конструкцию, которая часто более удобна, чем предлагаемая оператором while. Оператор for упрощает циклы, которые следуют общему шаблону. Большинство циклов имеют переменную счетчика какого-то вида. Такая переменная инициализируется до начала цикла и проверяется перед каждой итерацией цикла. Наконец, переменная счетчика инкрементируется или по-другому обновляется в конце тела цикла, прямо перед ее проверкой заново. В такой разновидности цикла инициализация, проверка и обновление являются тремя ключевыми манипуляциями переменной цикла. Оператор for кодирует каждую из этих трех манипуляций как выражение и делает такие выражения явной частью синтаксиса цикла:

```
for(инициализация ; проверка ; инкрементирование)
   оператор
```

инициализация, *проверка* и *инкрементирование* — три выражения (разделенные точками с запятой), которые отвечают за инициализацию, проверку и инкрементирование переменной цикла. Размещение их всех в первой строке цикла облегчает понимание того, что делает цикл `for`, и предупреждает такие ошибки, как забывание инициализировать или инкрементировать переменную цикла.

Объяснить, как работает цикл `for`, проще всего, показав эквивалентный цикл `while`[2]:

```
инициализация;
while(проверка) {
   оператор
   инкрементирование;
}
```

Другими словами, выражение *инициализация* вычисляется один раз перед началом цикла. Чтобы быть полезным, это выражение должно иметь побочные эффекты (обычно присваивание). JavaScript также разрешает *инициализации* быть оператором объявления переменных, а потому вы можете одновременно объявить и инициализировать счетчик цикла. Выражение *проверка* вычисляется перед каждой итерацией и управляет тем, будет ли выполняться тело цикла. Если результатом вычисления выражения *проверка* оказывается истинное значение, то *оператор*, являющийся телом цикла, выполняется. В заключение вычисляется выражение *инкрементирование*. Опять-таки, чтобы быть полезным, оно обязано быть выражением с побочными эффектами. Как правило, выражение *инкрементирование* либо представляет собой выражение присваивания, либо задействует операции `++` или `--`.

Мы можем вывести числа от 0 до 9 с помощью цикла `for` посредством приведенного далее кода. Сравните его с эквивалентным циклом `while` из предыдущего раздела:

```
for(let count = 0; count < 10; count++) {
   console.log(count);
}
```

Конечно, циклы могут становиться гораздо более сложными, нежели подобный простой пример, и временами на каждой итерации цикла изменяется множество переменных. Такая ситуация — единственное место, где обычно используется операция "запятая" в коде JavaScript; она позволяет объединить множество выражений инициализации и инкрементирования в одиночное выражение, подходящее для применения в цикле `for`:

```
let i, j, sum = 0;
for(i = 0, j = 10 ; i < 10 ; i++, j--) {
   sum += i * j;
}
```

[2] При обсуждении оператора `continue` в подразделе 5.5.3 мы увидим, что такой цикл `while` не является точным эквивалентом цикла `for`.

Во всех рассмотренных ранее примерах циклов переменная цикла была числовой. Так происходит часто, но вовсе не обязательно. В следующем коде цикл `for` используется для обхода структуры данных типа связного списка и возвращения последнего объекта в списке (т.е. первого объекта, который не имеет свойства `next`):

```
function tail(o) {  // Возвращает хвост связного списка o
   for(; o.next; o = o.next) /* пустое тело */ ; // Обход, пока свойство
                                                 // o.next истинное
   return o;
}
```

Обратите внимание, что в показанном цикле отсутствует выражение *инициализация*. В цикле `for` можно опускать любое из трех выражений, но две точки с запятой обязательны. Если вы опустите выражение *проверка*, тогда цикл будет повторяться нескончаемо долгий период времени, и `for(;;)` — еще один способ записи бесконечного цикла, подобный `while(true)`.

5.4.4. for/of

В ES6 определен новый оператор цикла: `for/of`. В новом цикле применяется ключевое слово `for`, но он представляет собой совершенно другой вид цикла в сравнении с обыкновенным циклом `for`. (Он также полностью отличается от более старого цикла `for/in`, который будет описан в подразделе 5.4.5.)

Цикл `for/of` работает с *итерируемыми* объектами. Мы объясним, что для объекта означает быть итерируемым, в главе 12, а пока достаточно знать, что массивы, строки, множества и отображения являются итерируемыми: они представляют последовательность или множество элементов, по которым можно проходить в цикле или выполнять итерацию посредством цикла `for/of`.

Например, вот как использовать `for/of` для циклического прохода по элементам массива чисел и подсчета их суммы:

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9], sum = 0;
for(let element of data) {
   sum += element;
}
sum    // => 45
```

Внешне синтаксис выглядит похожим на обыкновенный цикл `for`: ключевое слово `for`, за которым находятся круглые скобки, содержащие детали о том, что должен делать цикл. В данном случае круглые скобки содержат объявление переменной (или для уже объявленной переменной просто ее имя), далее следует ключевое слово `of` и выражение, результатом вычисления которого будет итерируемый объект, в этом случае массив `data`. Как и во всех циклах, затем идет тело цикла `for/of`, обычно в фигурных скобках.

В только что показанном коде тело цикла запускается по одному разу для каждого элемента массива `data`. Перед каждым выполнением тела цикла переменной `element` присваивается очередной элемент массива. Проход по элементам массива осуществляется с первого до последнего.

Итерация по массивам производится "вживую" — изменения, внесенные во время итерации, могут влиять на ее исход. Если мы модифицируем предыдущий код, добавив внутрь тела цикла строку `data.push(sum);`, то создадим бесконечный цикл, т.к. итерация никогда не доберется до последнего элемента массива.

for/of с объектами

Объекты (по умолчанию) не являются итерируемыми. Попытка применения `for/of` с обыкновенным объектом приводит к генерированию ошибки `TypeError` во время выполнения:

```
let o = { x: 1, y: 2, z: 3 };
for(let element of o) {  // Генерируется TypeError, потому что o -
                         // не итерируемый объект
   console.log(element);
}
```

Если вы хотите проходить по свойствам объекта, то можете использовать цикл `for/in` (рассматривается в подразделе 5.4.5) или `for/of` с методом `Object.keys()`:

```
let o = { x: 1, y: 2, z: 3 };
let keys = "";
for(let k of Object.keys(o)) {
   keys += k;
}
keys       // => "xyz"
```

Прием работает, поскольку метод `Object.keys()` возвращает массив имен свойств для объекта, а массивы допускают итерацию с помощью `for/of`. Также имейте в виду, что итерация по ключам объекта не выполняется вживую, как было ранее в примере с массивом — изменения, внесенные в объект `o` внутри тела цикла, не оказывают никакого влияния на итерацию. Если вас не интересуют ключи объекта, тогда можете организовать проход по значениям, соответствующим ключам:

```
let sum = 0;
for(let v of Object.values(o)) {
   sum += v;
}
sum        // => 6
```

А если вам нужны и ключи, и значения свойств объекта, то можете применить цикл `for/of` с методом `Object.entries()` и деструктурирующее присваивание:

```
let pairs = "";
for(let [k, v] of Object.entries(o)) {
   pairs += k + v;
}
pairs      // => "x1y2z3"
```

Метод `Object.entries()` возвращает массив массивов, где каждый внутренний массив представляет пару "ключ/значение" для одного свойства объекта. Мы используем в примере кода деструктурирующее присваивание для распаковки таких внутренних массивов в две индивидуальные переменные.

`for/of` со строками

Строки в ES6 итерируемы по символам:

```
let frequency = {};
for(let letter of "mississippi") {
   if (frequency[letter]) {
      frequency[letter]++;
   } else {
      frequency[letter] = 1;
   }
}
frequency      // => {m: 1, i: 4, s: 4, p: 2}
```

Обратите внимание, что итерация в строках производится по кодовым точкам Unicode, а не по символам UTF-16. Строка "I❤️🎉" имеет значение свойства `.length`, равное 5 (из-за того, что каждый из двух эмотиконов требует для своего представления двух символов UTF-16). Но в случае итерации по этой строке посредством `for/of` тело цикла будет выполнено три раза, по одному для каждой кодовой точки "I", "❤️" и "🎉".

`for/of` с классами `Set` и `Map`

Встроенные классы `Set` и `Map` в ES6 являются итерируемыми. При итерации по `Set` с помощью `for/of` тело цикла выполняется однократно для каждого элемента множества. Для вывода уникальных слов из строки текста вы могли бы применить следующий код:

```
let text = "Na na na na na na na na Batman!";
let wordSet = new Set(text.split(" "));
let unique = [];
for(let word of wordSet) {
   unique.push(word);
}
unique         // => ["Na", "na", "Batman!"]
```

Интересным случаем оказываются отображения, т.к. итерация объекта `Map` происходит не по ключам или по значениям, но по парам "ключ/значение". Каждый раз во время итерации итератор возвращает массив, первый элемент которого содержит ключ, а второй — соответствующее значение. Имея объект `Map` по имени `m`, вот как можно было бы выполнить итерацию с деструктуризацией его пар "ключ/значение":

```
let m = new Map([[1, "one"]]);
for(let [key, value] of m) {
   key         // => 1
   value       // => "one"
}
```

Асинхронная итерация с помощью for/await

В ES2018 вводится новый вид итератора, называемый *асинхронным итератором*, и разновидность цикла for/of, известная как цикл for/await, который работает с асинхронными итераторами.

Чтобы понять цикл for/await, вам необходимо ознакомиться с материалом глав 12 и 13, но ниже показано, как он выглядит:

```
// Читать порции из асинхронно итерируемого потока данных и выводить их
async function printStream(stream) {
   for await (let chunk of stream) {
      console.log(chunk);
   }
}
```

5.4.5. for/in

Цикл for/in очень похож на цикл for/of, в котором ключевое слово of заменено на in. В то время как цикл for/of требует итерируемого объекта после of, цикл for/in работает с любым объектом после in. Цикл for/of появился в ES6, но for/in с самого начала был частью языка JavaScript (вот почему он имеет более естественный синтаксис).

Оператор for/in организует цикл по именам свойств указанного объекта. Синтаксис выглядит следующим образом:

```
for (переменная in объект)
   оператор
```

Обычно на месте *переменной* указывается имя переменной, но здесь может быть объявление переменной или что-нибудь подходящее для левой стороны выражения присваивания. Кроме того, *объект* — это выражение, результатом вычисления которого будет объект. Как всегда, *оператор* представляет собой оператор или операторный блок, служащий телом цикла.

Вот как можно использовать цикл for/in:

```
for(let p in o) {     // Присвоить имена свойств объекта o переменной p
  console.log(o[p]); // Вывести значение каждого свойства
}
```

Чтобы выполнить оператор for/in, интерпретатор JavaScript сначала вычисляет выражение *объект*. Если результатом вычисления оказывается null или undefined, то интерпретатор пропускает цикл и переходит к следующему оператору. Далее интерпретатор выполняет тело цикла по одному разу для каждого перечислимого свойства объекта. Однако перед каждой итерацией интерпретатор вычисляет выражение *переменная* и присваивает ему имя свойства (строковое значение).

Обратите внимание, что *переменная* в цикле for/in может быть произвольным выражением при условии, что результатом ее вычисления является что-то подходящее для левой стороны присваивания. Выражение вычисляется при каждом проходе цикла, т.е. каждый раз оно может иметь разное значение.

Скажем, для копирования всех свойств объекта в массив вы можете применить такой код:

```
let o = { x: 1, y: 2, z: 3 };
let a = [], i = 0;
for(a[i++] in o) /* пустое тело */;
```

Массивы JavaScript — просто специализированный вид объектов, а индексы массива представляют собой свойства, которые можно перечислять с помощью цикла `for/in`. Например, следующая строка, дополняющая предыдущий код, перечисляет индексы массива 0, 1 и 2:

```
for(let i in a) console.log(i);
```

Я обнаружил, что распространенным источником ошибок в моем коде оказывается случайное использование с массивами цикла `for/in`, когда подразумевался `for/of`. При работе с массивами почти всегда нужно применять `for/of`, а не `for/in`.

Цикл `for/in` в действительности не перечисляет все свойства объекта. Он не перечисляет свойства, именами которых являются значения `Symbol`. К тому же среди свойств со строковыми именами цикл `for/in` проходит только по *перечислимым* свойствам (см. раздел 14.1). Различные встроенные методы, определяемые базовым JavaScript, не перечислимы. Скажем, все объекты имеют метод `toString()`, но цикл `for/in` не перечисляет это свойство `toString`. Кроме встроенных методов перечисление не поддерживают многие другие свойства встроенных объектов. Все свойства и методы, определяемые вашим кодом, по умолчанию будут перечислимыми. (Вы можете сделать их не перечислимыми с использованием методик, которые объясняются в разделе 14.1.)

Перечислимые унаследованные свойства (см. подраздел 6.3.2) также перечисляются циклом `for/in`. Это означает, что если вы применяете циклы `for/in` и также используете код, где определяются объекты, которые наследуются всеми объектами, тогда ваши циклы могут вести себя не так, как ожидалось. По указанной причине многие программисты вместо цикла `for/in` предпочитают применять цикл `for/of` с `Object.keys()`.

Если в теле цикла `for/in` удаляется свойство, которое еще не перечислялось, то оно перечисляться не будет. Если в теле цикла определяются новые свойства объекта, то они могут как перечисляться, так и не перечисляться. Дополнительные сведения о порядке перечисления свойств циклом `for/in` ищите в подразделе 6.6.1.

5.5. Переходы

Еще одной категорией операторов JavaScript являются *операторы переходов*. Как вытекает из их названия, они заставляют интерпретатор JavaScript переходить в новое место внутри исходного кода. Оператор `break` вынуждает интерпретатор перейти в конец цикла или другого оператора. Оператор `continue` заставляет интерпретатор пропустить остаток тела цикла и перейти в начало цикла, чтобы начать новую итерацию. JavaScript разрешает операторам быть

именованными, или *помеченными*, а `break` и `continue` способны идентифицировать целевую метку цикла или другого оператора.

Оператор `return` вынуждает интерпретатор выйти из вызова функции обратно в код, который ее вызвал, и также предоставляет значение для вызова. Оператор `throw` является своего рода промежуточным возвратом из генераторной функции. Оператор `throw` *инициирует* исключение и предназначен для работы с оператором `try/catch/finally`, который устанавливает блок кода обработки исключений. Это сложный вид оператора перехода: когда исключение инициировано, интерпретатор переходит к ближайшему включающему обработчику исключений, который может находиться в той же функции или внутри вызывающей функции выше в стеке вызовов.

Детали о каждом операторе перехода приводятся в последующих разделах.

5.5.1. Помеченные операторы

Любой оператор может быть *помечен* путем его предварения идентификатором и двоеточием:

идентификатор: *оператор*

Помечая оператор, вы даете ему имя, которое можно использовать для ссылки на него из какого-то другого места в программе. Вы можете пометить любой оператор, хотя полезно помечать только операторы, имеющие тела, такие как циклы и условные операторы. За счет предоставления циклу имени вы можете применять внутри его тела операторы `break` и `continue` для выхода из цикла или перехода прямо в начало цикла, чтобы начать новую итерацию. В JavaScript только операторы `break` и `continue` используют метки; они раскрываются в последующих подразделах. Ниже показан пример помеченного цикла `while` и оператора `continue`, который задействует метку:

```
mainloop: while(token !== null) {
    // Код опущен...
    continue mainloop; //Перейти к следующей итерации именованного цикла
    // Код опущен...
}
```

Применяемый для пометки оператора *идентификатор* может быть любым допустимым идентификатором JavaScript, который не является зарезервированным словом. Пространство имен для меток не совпадает с пространством имен для переменных и функций, поэтому вы можете использовать один и тот же идентификатор в качестве метки оператора и имени переменной или функции. Метки операторов определены только внутри операторов, к которым они применяются (и, разумеется, внутри их подоператоров). Оператор может не иметь такую же метку, как содержащий его оператор, но два оператора могут иметь одну и ту же метку при условии, что они не вложены друг в друга. Помеченные операторы можно помечать снова. Фактически это означает, что любой оператор может иметь множество меток.

5.5.2. break

Оператор `break`, используемый в одиночку, обеспечивает немедленное завершение самого внутреннего включающего оператора цикла или `switch`. Его синтаксис прост:

```
break;
```

Поскольку происходит выход из цикла или `switch`, такая форма оператора `break` разрешена только внутри указанных операторов.

Вы уже встречали примеры оператора `break` внутри оператора `switch`. В циклах он обычно применяется для преждевременного выхода, когда по какой-то причине больше нет необходимости продолжать цикл до полного завершения. Если с циклом связаны сложные условия окончания, то часто легче реализовать ряд таких условий с помощью операторов `break`, а не пытаться выразить их все в единственном выражении цикла. В следующем коде производится поиск элементов массива с определенным значением. Цикл заканчивается нормально, когда достигается конец массива, и завершается посредством `break`, если удается найти то, что ищется в массиве:

```
for(let i = 0; i < a.length; i++) {
   if (a[i] === target) break;
}
```

JavaScript также разрешает дополнять ключевое слово `break` меткой оператора (просто идентификатором без двоеточия):

```
break имя_метки;
```

Когда оператор `break` используется с меткой, он вызывает переход в конец, или прекращение, включающего оператора, который имеет указанную метку. Если эта форма `break` применяется в отсутствие включающего оператора с указанной меткой, то возникает синтаксическая ошибка. При такой форме оператора `break` именованный оператор не обязательно должен быть циклом или `switch`: `break` может "выйти" из любого включающего оператора. Он может быть даже операторным блоком, сгруппированным внутри фигурных скобок с единственной целью — снабжение блока меткой.

Использовать символ новой строки между ключевым словом `break` и *именем_метки* не разрешено. Причина связана с принятой в JavaScript автоматической вставкой пропущенных точек с запятой. Если вы поместите разделитель строк между ключевым словом `break` и следующей за ним меткой, тогда интерпретатор JavaScript предполагает, что вы намеревались применять простую непомеченную форму оператора, и трактует разделитель строк как точку с запятой. (См. раздел 2.6.)

Помеченная форма оператора `break` нужна, когда вы хотите выйти из оператора, который не является ближайшим включающим циклом или `switch`, что демонстрируется в коде ниже:

```
let matrix = getData();  // Получить откуда-то двумерный массив чисел
// Просуммировать все числа в матрице
let sum = 0, success = false;
```

```
// Начать с помеченного оператора, из которого можно выйти
// в случае возникновения ошибки
computeSum: if (matrix) {
   for(let x = 0; x < matrix.length; x++) {
      let row = matrix[x];
      if (!row) break computeSum;
      for(let y = 0; y < row.length; y++) {
         let cell = row[y];
         if (isNaN(cell)) break computeSum;
         sum += cell;
      }
   }
   success = true;
}
// Операторы break переходят сюда. Если мы оказываемся здесь
// с success == false, тогда с матрицей что-то пошло не так.
// В противном случае sum содержит сумму всех ячеек матрицы.
```

В заключение отметим, что оператор `break` с меткой или без нее не способен передавать управление между границами функций. Скажем, вы не можете помечать оператор определения функции и затем использовать эту метку внутри функции.

5.5.3. continue

Оператор `continue` похож на `break`. Тем не менее, вместо выхода из цикла `continue` перезапускает цикл со следующей итерации. Синтаксис оператора `continue` в той же мере прост, как синтаксис `break`:

```
continue;
```

Оператор `continue` также может применяться с меткой:

```
continue имя_метки;
```

В помеченной и непомеченной формах оператор `continue` может использоваться только внутри тела цикла. Его появление в любом другом месте вызывает синтаксическую ошибку. Когда оператор `continue` выполняется, текущая итерация включающего цикла прекращается, а следующая начинается. Для разных видов циклов это означает разные вещи.

- В цикле `while` указанное в операторе цикла выражение проверяется снова, и если оно истинно, тогда тело цикла выполняется с самого начала.
- В цикле `do/while` управление переходит в конец цикла, где снова проверяется условие цикла, прежде чем начинать цикл с начала.
- В цикле `for` вычисляется выражение инкрементирование и затем снова проверяется выражение проверка, чтобы выяснить, должна ли выполняться еще одна итерация.
- В цикле `for/of` или `for/in` цикл начинается заново со следующего итерируемого значения или следующего имени свойства, присваиваемого указанной переменной.

Обратите внимание на отличие в поведении оператора continue в циклах while и for: цикл while возвращается прямо к своему условию, но цикл for сначала вычисляет выражение *инкрементирование* и затем возвращается к своему условию. Ранее мы рассматривали поведение цикла for в терминах "эквивалентного" цикла while. Однако из-за того, что оператор continue ведет себя по-разному в этих двух циклах, точно эмулировать цикл for посредством одного лишь while на самом деле невозможно.

В следующем примере непомеченный оператор continue применяется для пропуска остатка текущей итерации цикла, когда возникает ошибка:

```
for(let i = 0; i < data.length; i++) {
    if (!data[i]) continue;  // Не удается продолжить
                             // с неопределенными данными
    total += data[i];
}
```

Как и break, оператор continue может использоваться в своей помеченной форме внутри вложенных циклов, когда подлежащий перезапуску цикл не является непосредственно включающим. Кроме того, как и в операторе break, разрывы строк между ключевым словом continue и *именем_метки* не разрешены.

5.5.4. return

Вспомните, что вызовы функций являются выражениями, а все выражения имеют значения. Оператор return внутри функции указывает значение вызовов этой функции. Вот синтаксис оператора return:

```
return выражение;
```

Оператор return может встречаться только внутри тела функции. Его появление в любом другом месте будет синтаксической ошибкой. Выполнение оператора return приводит к тому, что содержащая его функция возвращает значение *выражения* вызывающему коду. Например:

```
function square(x) { return x*x; }  // Функция, которая имеет
                                    // оператор return
square(2)                           // => 4
```

Без оператора return вызов функции просто обеспечивает выполнение всех операторов в теле функции по очереди, пока не будет достигнут конец функции, после чего управление возвращается вызывающему коду. В таком случае выражение вызова вычисляется как undefined. Оператор return часто оказывается последним оператором в функции, но он не обязан быть таковым: функция возвращает управление вызывающему коду, когда выполняется return, даже если в теле функции остались другие операторы.

Оператор return может также применяться без *выражения*, чтобы функция возвращала undefined вызывающему коду. Вот пример:

```
function displayObject(o) {
    // Немедленный возврат, если аргумент равен null или undefined
```

```
    if (!o) return;
    // Остальной код функции...
}
```

Из-за автоматической вставки точек с запятой JavaScript (см. раздел 2.6) вы не можете помещать разрыв строки между ключевым словом `return` и следующим за ним выражением.

5.5.5. yield

Оператор `yield` во многом похож на `return`, но используется только в генераторных функциях ES6 (см. раздел 12.3) для выпуска следующего значения в генерируемой последовательности значений без фактического возврата управления:

```
// Генераторная функция, которая выдает диапазон целых чисел
function* range(from, to) {
    for(let i = from; i <= to; i++) {
        yield i;
    }
}
```

Чтобы понять `yield`, нужно освоить итераторы и генераторы, которые будут обсуждаться в главе 12. Тем не менее, оператор `yield` включен здесь ради полноты. (Однако, как объясняется в подразделе 12.4.2, формально `yield` является операцией, а не оператором.)

5.5.6. throw

Исключение представляет собой сигнал, который указывает, что возникло какое-то необычное условие или ошибка. *Генерация* исключения означает предупреждение о таком ошибочном или необычном условии. *Перехват* исключения означает его обработку — выполнение любых действий, необходимых или подходящих для восстановления после исключения. Исключения в JavaScript генерируются всякий раз, когда происходит ошибка во время выполнения или программа явно генерирует исключение с применением оператора `throw`. Исключения перехватываются с помощью оператора `try/catch/finally`, который будет описан в следующем разделе.

Оператор `throw` имеет следующий синтаксис:

```
throw выражение;
```

Результатом вычисления *выражения* может быть значение любого типа. В `throw` можно указывать число, представляющее код ошибки, или строку, которая содержит понятное человеку сообщение об ошибке. Когда интерпретатор JavaScript самостоятельно генерирует ошибку, используется класс `Error` и его подклассы, но вы тоже можете их применять. Объект `Error` имеет свойство `name`, указывающее тип ошибки, и свойство `message`, которое хранит строку, переданную функции конструктора. Ниже приведен пример функции, которая генерирует объект `Error` в случае вызова с недопустимым аргументом:

```
function factorial(x) {
    // Если входной аргумент не является допустимым,
    // то сгенерировать исключение!
    if (x < 0) throw new Error("Значение x не должно быть отрицательным");
    // Иначе вычислить и возвратить значение обычным образом
    let f;
    for(f = 1; x > 1; f *= x, x--) /* пустое тело */ ;
    return f;
}
factorial(4)    // => 24
```

Когда сгенерировано исключение, интерпретатор JavaScript немедленно останавливает нормальное выполнение программы и переходит к ближайшему обработчику исключений. Обработчики исключений записываются с использованием конструкции `catch` оператора `try/catch/finally`, который описан в следующем разделе. Если блок кода, в котором было сгенерировано исключение, не имеет ассоциированной конструкции `catch`, тогда интерпретатор проверяет ближайший включающий блок кода на предмет наличия в нем ассоциированного обработчика исключений. Процесс продолжается до тех пор, пока обработчик не будет найден. Если исключение было сгенерировано в функции, которая не содержит оператор `try/catch/finally` для его обработки, то исключение распространяется в код, вызвавший функцию. Таким образом, исключения распространяются вверх по лексической структуре методов JavaScript и вверх по стеку вызовов. Если в итоге обработчик исключений не был найден, тогда исключение трактуется как ошибка и о ней сообщается пользователю.

5.5.7. try/catch/finally

Оператор `try/catch/finally` является механизмом обработки исключений JavaScript. В конструкции `try` данного оператора просто определяется блок кода, чьи исключения должны быть обработаны. За блоком `try` следует конструкция `catch`, которая представляет собой блок операторов, вызываемый при возникновении исключения где-нибудь внутри блока `try`. После конструкции `catch` находится блок `finally`, содержащий код очистки, который гарантированно выполнится вне зависимости от того, что произошло в блоке `try`. Блоки `catch` и `finally` необязательны, но блок `try` должен сопровождаться, по крайней мере, одним из этих блоков. Блоки `try`, `catch` и `finally` начинаются и заканчиваются фигурными скобками, которые являются обязательной частью синтаксиса и не могут быть опущены, даже если конструкция содержит только один оператор.

В следующем коде иллюстрируется синтаксис и предназначение оператора `try/catch/finally`:

```
try {
    // В нормальной ситуации этот код выполняется от начала до конца блока
    // безо всяких проблем. Но иногда он может генерировать исключение,
    // либо напрямую с помощью оператора throw, либо косвенно за счет вызова
    // метода, который генерирует исключение.
}
```

```
catch(e) {
  // Операторы в данном блоке выполняются, если и только если в блоке try
  // было сгенерировано исключение. Эти операторы могут использовать
  // локальную переменную e для ссылки на объект Error или другое значение,
  // которое было указано в throw. В блоке можно каким-то образом
  // обработать исключение, проигнорировать его, ничего не делая,
  // или повторно сгенерировать исключение с помощью throw.
}

finally {
  // Данный блок содержит операторы, которые всегда выполняются
  // независимо от того, что произошло в блоке try.
  // Они выполняются при завершении блока try:
  // 1) нормальным образом после того, как достигнут конец блока;
  // 2) из-за оператора break, continue или return;
  // 3) из-за исключения, которое было обработано конструкцией catch выше;
  // 4) из-за необработанного исключения, которое продолжило
  //    свое распространение.
}
```

Обратите внимание, что обычно за ключевым словом `catch` следует идентификатор в круглых скобках, который подобен параметру функции. Когда исключение перехватывается, ассоциированное с ним значение (скажем, объект `Error`) присваивается этому параметру. Идентификатор в конструкции `catch` имеет блочную область видимости — он определен только в пределах блока `catch`.

Ниже показан реалистичный пример оператора `try/catch`. В нем применяется метод `factorial()`, определенный в предыдущем разделе, и методы JavaScript стороны клиента `prompt()` и `alert()` для ввода и вывода:

```
try {
   // Запросить у пользователя ввод числа
   let n = Number(prompt("Введите положительное число", ""));

   // Вычислить факториал числа, предполагая, что введенное число допустимо
   let f = factorial(n);

   // Вывести результат
   alert(n + "! = " + f);
}
catch(ex) {    // Если введенное пользователем число было
               // недопустимым, то мы оказываемся здесь
   alert(ex); // Сообщить пользователю об ошибке
}
```

В примере приведен оператор `try/catch` без конструкции `finally`. Хотя конструкция `finally` не используется настолько часто, как `catch`, она может быть полезной. Тем не менее, ее поведение требует дополнительного пояснения. Конструкция `finally` гарантированно выполняется в случае выполнения любой порции блока `try` вне зависимости от того, каким образом завершил свою работу код в блоке `try`. Обычно она применяется для очистки после выполнения кода в конструкции `try`.

В нормальной ситуации интерпретатор JavaScript достигает конца блока try и затем переходит к блоку finally, который выполняет любую необходимую очистку. Если интерпретатор покидает блок try из-за оператора return, continue или break, то блок finally выполняется до того, как интерпретатор перейдет в новое место назначения.

Если в блоке try возникает исключение и есть ассоциированный блок catch для его обработки, тогда интерпретатор сначала выполняет блок catch и затем блок finally. Если же локальный блок catch для обработки исключения отсутствует, то интерпретатор сначала выполняет блок finally, после чего переходит к ближайшей внешней конструкции catch.

Если сам блок finally инициирует переход посредством оператора return, continue, break или throw либо за счет вызова метода, который генерирует исключение, тогда интерпретатор отменяет любой незаконченный переход и выполняет новый переход. Например, если в конструкции finally генерируется исключение, то оно замещает любое исключение, которые было в процессе генерации. Если в конструкции finally выдается оператор return, то происходит нормальный возврат из метода, даже когда было сгенерировано исключение, которое еще не обработано.

Конструкции try и finally могут использоваться вместе без catch. В таком случае блок finally является просто кодом очистки, который гарантированно выполняется независимо от того, что произошло в блоке try. Вспомните, что мы не можем в полной мере эмулировать цикл for с помощью цикла while, поскольку оператор continue в этих двух циклах ведет себя по-разному. Если мы добавим оператор try/finally, то можем записать цикл while, который работает подобно циклу for и корректно обрабатывает операторы continue:

```
// Эмуляция for(инициализация ; проверка ; инкрементирование) тело;
инициализация ;
while( проверка ) {
    try { тело ; }
    finally { инкрементирование ; }
}
```

Однако обратите внимание, что *тело*, которое содержит оператор break, в цикле while ведет себя слегка иначе (вызывая дополнительное инкрементирование перед выходом) по сравнению с циклом for, поэтому даже посредством конструкции finally точная эмуляция цикла for с помощью while невозможна.

Пустые конструкции catch

Иногда вы можете обнаружить, что применяете конструкцию catch единственно для того, чтобы выявлять и прекращать распространение исключения, даже если тип или значение исключения вас не интересует. В ES2019 и последующих версиях вы можете опустить круглые скобки и идентификатор и в таком случае использовать только одно ключевое слово catch. Вот пример:

```
// Похожа на JSON.parse(), но вместо генерации ошибки возвращает undefined
function parseJSON(s) {
    try {
        return JSON.parse(s);
    } catch {
        // Что-то пошло не так, но нас не интересует, что именно
        return undefined;
    }
}
```

5.6. Смешанные операторы

В этом разделе описаны три оставшихся оператора JavaScript — `with`, `debugger` и `"use strict"`.

5.6.1. `with`

Оператор `with` выполняет блок кода, как если бы свойства указанного объекта были переменными в области видимости данного кода. Он имеет следующий синтаксис:

```
with (объект)
    оператор
```

Этот оператор создает временную область видимости со свойствами *объекта* в качестве переменных и затем выполняет *оператор* внутри такой области видимости.

Оператор `with` запрещен в строгом режиме (см. подраздел 5.6.3) и должен восприниматься как нерекомендуемый в нестрогом режиме: по возможности избегайте его применения. Код JavaScript, в котором используется `with`, трудно поддается оптимизации и, вероятно, будет выполняться значительно медленнее, чем эквивалентный код, написанный без оператора `with`.

Обычно оператор `with` применяется для облегчения работы с глубоко вложенными иерархиями объектов. Например, в коде JavaScript на стороне клиента вам может понадобиться набирать выражения вроде показанного ниже, чтобы получать доступ к элементам HTML-формы:

```
document.forms[0].address.value
```

Если необходимо многократно записывать выражения подобного рода, тогда можете использовать оператор `with` для обращения со свойствами объекта формы как с переменными:

```
with(document.forms[0]) {
    // Далее появляется возможность прямого доступа к элементам формы,
    // например:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

В итоге уменьшается количество набираемых символов: вам больше не нужно предварять имя каждого свойства формы строкой `document.forms[0]`. Конечно, столь же просто можно избежать применения оператора `with` и записать предыдущий код в следующем виде:

```
let f = document.forms[0];
f.name.value = "";
f.address.value = "";
f.email.value = "";
```

Обратите внимание, что если вы используете `const`, `let` или `var` для объявления переменной или константы внутри тела оператора `with`, то тем самым создаете обыкновенную переменную, а вовсе не определяете новое свойство в указанном объекте.

5.6.2. debugger

Оператор `debugger` обычно ничего не делает. Тем не менее, если программа отладчика доступна и работает, то реализация может (но не обязана) предпринимать определенные действия отладки. На практике оператор `debugger` действует подобно точке останова: выполнение кода JavaScript останавливается, и вы можете применять отладчик для вывода значений переменных, просмотра стека вызовов и т.д. Предположим, например, что вы получили исключение в своей функции `f()` из-за ее вызова с аргументом `undefined`, но не в состоянии выяснить, где такой вызов произошел. Чтобы помочь в поиске источника проблемы, вы могли бы изменить функцию `f()`, начав ее так:

```
function f(o) {
  if (o === undefined) debugger; //Временная строка для отладочных целей
  ...                            //Далее идет остальной код функции
}
```

Теперь, когда `f()` вызывается без аргументов, выполнение остановится, а вы сможете использовать отладчик для инспектирования стека вызовов и выяснения, откуда поступает некорректный вызов. Следует отметить, что наличия отладчика недостаточно: оператор `debugger` не запускает отладчик. Однако если вы работаете с веб-браузером и в нем открыта консоль инструментов разработчика, то оператор `debugger` послужит точкой останова.

5.6.3. "use strict"

`"use strict"` — это директива, появившаяся в ES5. Директивы не являются операторами (но достаточно близки, чтобы документировать `"use strict"` здесь). Существуют два важных отличия между директивой `"use strict"` и обыкновенными операторами.

- Она не содержит любые ключевые слова языка: директива представляет собой просто оператор выражения, состоящий из специального строкового литерала (в одинарных или двойных кавычках).
- Она может находиться только в начале сценария или тела функции до появления каких-либо подлинных операторов.

Директива "use strict" предназначена для указания, что следующий за ней код (в сценарии или функции) будет *строгим кодом*. Код верхнего уровня (не внутри функции) сценария является строгим кодом, если в сценарии имеется директива "use strict". Тело функции считается строгим кодом, если функция определена в строгом коде или имеет директиву "use strict". Код, передаваемый методу eval(), будет строгим кодом, если eval() вызывается из строгого кода или строка кода включает директиву "use strict". В дополнение к коду, явно объявленному как строгий, любой код в теле класса (см. главу 9) или в модуле ES6 (см. раздел 10.3) автоматически становится строгим. Таким образом, если весь ваш код записан в виде модулей, тогда он весь автоматически будет строгим, и у вас никогда не возникнет необходимость явно применять директиву "use strict".

Строгий код выполняется в *строгом режиме*. Строгий режим представляет собой ограниченное подмножество языка, в котором устранены важные языковые недостатки, а также обеспечены более строгая проверка ошибок и увеличенная безопасность. Поскольку строгий режим не выбирается по умолчанию, старый код JavaScript, в котором используются дефектные унаследованные средства языка, по-прежнему будет нормально работать. Ниже объясняются отличия между строгим и нестрогим режимами (особенно важны первые три).

- В строгом режиме оператор with не разрешен.
- В строгом режиме все переменные должны быть объявлены: присваивание значения идентификатору, который не является объявленной переменной, функцией, параметром функции, параметром конструкции catch или свойством глобального объекта, приводит к генерации ReferenceError. (В нестрогом режиме просто неявно объявляется глобальная переменная путем добавления нового свойства к глобальному объекту.)
- В строгом режиме функции, вызываемые как функции (а не как методы), имеют значение this, равное undefined. (В нестрогом режиме функциям, вызываемым как функции, в качестве их значения this всегда передается глобальный объект.) Кроме того, в строгом режиме при вызове функции с помощью call() или apply() (см. подраздел 8.7.4) значением this будет именно то значение, которое передается как первый аргумент call() или apply(). (В нестрогом режиме значения null и undefined заменяются глобальным объектом, а необъектные значения преобразуются в объекты.)
- В строгом режиме присваивания не поддерживающим запись свойствам и попытки создания новых свойств в не допускающих расширение объектах приводят к генерации ошибки TypeError. (В нестрогом режиме такие попытки молча терпят неудачу.)
- В строгом режиме коду, передаваемому eval(), не удастся объявлять переменные или определять функции в области видимости вызывающего кода, как это можно делать в нестрогом режиме. Взамен определения переменных и функций помещаются в новую область видимости, созданную для eval(). После возврата управления из eval() эта область видимости отбрасывается.

- В строгом режиме объект `arguments` (см. подраздел 8.3.3) в функции хранит статическую копию значений, переданных функции. В нестрогом режиме объект `arguments` обладает "магическим" поведением, при котором элементы массива и именованные параметры функции ссылаются на то же самое значение.
- В строгом режиме генерируется ошибка `SyntaxError`, если за операцией `delete` следует неуточненный идентификатор, такой как переменная, функция или параметр функции. (В нестрогом режиме подобного рода выражение `delete` ничего не делает и вычисляется как `false`.)
- В строгом режиме попытка удаления неконфигурируемого свойства приводит к генерации ошибки `TypeError`. (В нестрогом режиме такая попытка терпит неудачу, а выражение `delete` вычисляется как `false`.)
- В строгом режиме определение в объектном литерале двух или большего количества свойств с одинаковыми именами считается синтаксической ошибкой. (В нестрогом режиме ошибка не возникает.)
- В строгом режиме наличие в объявлении функции двух или большего количества параметров с одинаковыми именами считается синтаксической ошибкой. (В нестрогом режиме ошибка не возникает.)
- В строгом режиме восьмеричные целочисленные литералы (начинающиеся с 0, за которым не следует x) не разрешены. (В нестрогом режиме некоторые реализации допускают восьмеричные литералы.)
- В строгом режиме идентификаторы `eval` и `arguments` трактуются подобно ключевым словам, и вам не разрешено изменять их значение. Вы не можете присваивать значение этим идентификаторам, объявлять их как переменные, применять их в качестве имен функций, использовать их как имена параметров функций или применять в качестве идентификаторов блоков `catch`.
- В строгом режиме возможность просмотра стека вызовов ограничена. В функции строгого режима обращение к `arguments.caller` и `arguments.callee` приводит к генерации ошибки `TypeError`. Функции строгого режима также имеют свойства `caller` и `arguments`, чтение которых вызывает генерацию `TypeError`. (В некоторых реализациях такие нестандартные свойства определяются в нестрогих функциях.)

5.7. Объявления

Ключевые слова `const`, `let`, `var`, `function`, `class`, `import` и `export` формально не являются операторами, но они во многом похожи на операторы, и в книге они неформально относятся к операторам, поэтому указанные ключевые слова заслуживают упоминания в настоящей главе.

Такие ключевые слова более точно называть *объявлениями*, а не операторами. В начале этой главы говорилось о том, что операторы призваны "заставить, чтобы

что-то произошло". Объявления предназначены для определения новых значений и назначения им имен, которые можно использовать для ссылки на новые значения. Сами по себе они не особо много делают, чтобы что-то произошло, но за счет предоставления имен для значений обоснованно определяют смысл остальных операторов в программе.

Когда программа запускается, существуют выражения программы, которые вычисляются, и операторы программы, которые выполняются. Объявления в программе не "выполняются" тем же способом: взамен они определяют структуру самой программы. Грубо говоря, вы можете считать объявления частью программы, которая обрабатывается до того, как код начинает выполнение.

Объявления JavaScript применяются для определения констант, переменных, функций и классов, а также для импортирования и экспортирования значений между модулями. В последующих подразделах приводятся примеры объявлений подобного рода. Все они более детально рассматриваются в других местах книги.

5.7.1. const, let и var

Объявления `const`, `let` и `var` раскрывались в разделе 3.10. В ES6 и более поздних версиях `const` объявляет константы, а `let` — переменные. До выхода ES6 ключевое слово `var` было единственным способом объявления переменных, а для объявления констант вообще ничего не предлагалось. Переменные, объявленные с помощью `var`, имели область видимости в пределах содержащей функции, а не содержащего блока. Это могло быть источником ошибок и в современном языке JavaScript действительно нет никаких причин использовать `var` вместо `let`.

```
const TAU = 2*Math.PI;
let radius = 3;
var circumference = TAU * radius;
```

5.7.2. function

Объявление `function` применяется для определения функций, которые подробно рассматриваются в главе 8. (Ключевое слово `function` также встречалось в разделе 4.3, где оно использовалось как часть выражения определения функции, а не объявления функции.) Объявление функции выглядит следующим образом:

```
function area(radius) {
   return Math.PI * radius * radius;
}
```

Объявление функции создает объект функции и присваивает его указанному имени — `area` в приведенном примере. С применением такого имени мы можем ссылаться на функцию и выполнять код внутри нее из других мест программы. Объявления функций в любом блоке кода JavaScript обрабатываются до выполнения этого кода, а имена функций привязываются к объектам функций по всему блоку. Мы говорим, что объявления функций "поднимаются", т.к. они

будто бы перемещаются в верхнюю часть области видимости, внутри которой определены. В результате код, где функция вызывается, может располагаться в программе раньше кода, объявляющего функцию.

В разделе 12.3 описан особый вид функций — *генераторы*. Объявления генераторов используют ключевое слово `function`, но за ним следует звездочка. В разделе 13.3 обсуждаются асинхронные функции, которые тоже объявляются с применением ключевого слова `function`, но его предваряет ключевое слово `async`.

5.7.3. `class`

В ES6 и последующих версиях объявление `class` создает новый класс и назначает ему имя, посредством которого на класс можно ссылаться. Классы подробно рассматриваются в главе 9. Объявление простого класса может выглядеть так:

```
class Circle {
   constructor(radius) { this.r = radius; }
   area() { return Math.PI * this.r * this.r; }
   circumference() { return 2 * Math.PI * this.r; }
}
```

В отличие от функций объявления классов не поднимаются, поэтому объявленный таким способом класс нельзя использовать в коде, находящемся до объявления.

5.7.4. `import` и `export`

Объявления `import` и `export` применяются вместе, чтобы сделать значения, определенные в одном модуле кода JavaScript, доступными в другом модуле. Модуль представляет собой файл кода JavaScript, который имеет собственное глобальное пространство имен, полностью независимое от всех остальных модулей. Значение (такое как функция или класс), определенное в одном модуле, можно использовать в другом модуле, только если определяющий модуль экспортирует его с помощью `export`, а потребляющий модуль импортирует его посредством `import`. Модули обсуждаются в главе 10, а `import` и `export` — в разделе 10.3.

Директивы `import` применяются для импортирования одного и более значений из другого файла кода JavaScript и назначения им имен внутри текущего модуля. Директивы `import` принимают несколько разных форм. Ниже показаны примеры:

```
import Circle from './geometry/circle.js';
import { PI, TAU } from './geometry/constants.js';
import { magnitude as hypotenuse } from './vectors/utils.js';
```

Значения внутри модуля JavaScript являются закрытыми и не могут импортироваться в другие модули, если они явно не экспортированы. Это делает директива `export`: она указывает, что одно или большее количество значений, определенных в текущем модуле, экспортируются и потому будут доступны для импортирования другими модулями.

Директива export имеет больше вариантов, чем директива import. Вот один из них:

```
// geometry/constants.js
const PI = Math.PI;
const TAU = 2 * PI;
export { PI, TAU };
```

Ключевое слово export иногда используется как модификатор в другом объявлении, давая в результате составное объявление, которое определяет константу, переменную, функцию или класс и одновременно экспортирует его. И когда модуль экспортирует только одно значение, то это делается с помощью специальной формы export default:

```
export const TAU = 2 * Math.PI;
export function magnitude(x,y) { return Math.sqrt(x*x + y*y); }
export default class Circle { /* определение класса опущено */ }
```

5.8. Резюме по операторам JavaScript

В главе были представлены операторы языка JavaScript, сводка по которым приведена в табл. 5.1.

Таблица 5.1. Синтаксис операторов JavaScript

Оператор	Предназначение
break	Выходит из самого внутреннего цикла или switch либо из именованного включающего оператора
case	Помечает оператор внутри switch
class	Объявляет класс
const	Объявляет и инициализирует одну или большее количество констант
continue	Начинает следующую итерацию самого внутреннего цикла или именованного цикла
debugger	Точка останова отладчика
default	Помечает оператор по умолчанию внутри switch
do/while	Альтернатива циклу while
export	Объявляет значения, которые могут быть импортированы другими модулями
for	Легкий в применении цикл
for/await	Асинхронно проходит по значениям асинхронного итератора
for/in	Перечисляет имена свойств объекта
for/of	Перечисляет значения итерируемого объекта, такого как массив
function	Объявляет функцию
if/else	Выполняет один или другой оператор в зависимости от условия
import	Объявляет имена для значений, объявленных в других модулях

Оператор	Предназначение
метка	Назначает оператору имя для использования с `break` и `continue`
`let`	Объявляет и инициализирует одну или большее количество переменных с блочной областью видимости (новый синтаксис)
`return`	Возвращает значение из функции
`switch`	Реализует разветвление по множеству меток `case` или `default:`
`throw`	Генерирует исключение
`try/catch/finally`	Обрабатывает исключения и обеспечивает код очистки
`"use strict"`	Применяет ограничения строгого режима к сценарию или функции
`var`	Объявляет и инициализирует одну или большее количество переменных (старый синтаксис)
`while`	Базовая конструкция цикла
`with`	Расширяет цепочку областей видимости (не рекомендуется к использованию и запрещен в строгом режиме)
`yield`	Предоставляет значение, подлежащее итерации; применяется только в генераторных функциях

ГЛАВА 6

Объекты

Объекты являются наиболее фундаментальным типом данных JavaScript и вы уже видели их много раз в предшествующих главах. Поскольку объекты настолько существенны для языка JavaScript, важно понимать их работу в мельчайших деталях, которые и раскрываются в этой главе. Глава начинается с формального обзора объектов, после чего предлагаются практические разделы, посвященные созданию объектов и запрашиванию, установке, удалению, проверке и перечислению их свойств. За разделами, ориентированными на свойства, следуют разделы, где объясняется, как расширять, сериализировать и определять важные методы в объектах. Глава завершается крупным разделом о новом синтаксисе объектных литералов, введенном в ES6 и более современных версиях языка.

6.1. Введение в объекты

Объект — это составное значение: он агрегирует множество значений (элементарные значения или другие объекты) и позволяет хранить и извлекать внутренние значения по имени. Объект представляет собой неупорядоченную коллекцию *свойств*, каждое из которых имеет имя и значение. Имена свойств обычно являются строками (хотя, как будет показано в подразделе 6.10.3, имена свойств могут также быть значениями `Symbol`), а потому мы можем говорить, что объекты отображают строки на значения. Такое сопоставление строк со значениями называется по-разному: возможно, вы уже знакомы с фундаментальной структурой данных по имени "хеш", "хеш-таблица", "словарь" или "ассоциативный массив". Однако объект — нечто большее, чем простое отображение строк на значения. Помимо поддержки собственного набора свойств объект JavaScript также наследует свойства еще одного объекта, известного как его "прототип". Методы объекта обычно представляют собой наследуемые свойства, и такое "наследование прототипов" считается ключевой возможностью JavaScript.

Объекты JavaScript являются динамическими, т.к. свойства обычно можно добавлять и удалять, но они могут использоваться для эмуляции статических объектов и "структур" из статически типизированных языков. Их также можно применять (за счет игнорирования части, касающейся значений, в отображении строк на значения) для представления наборов строк.

В JavaScript любое значение, отличающееся от строки, числа, значения Symbol, true, false, null или undefined, является объектом. И хотя строки, числа и булевские значения не относятся к объектам, они могут вести себя как неизменяемые объекты.

Вспомните из раздела 3.8, что объекты изменяемы и обрабатываются по ссылке, а не по значению. Если переменная x ссылается на объект и выполнился код `let y = x;`, тогда переменная y сохранит ссылку на тот же самый объект, но не его копию. Любые изменения, внесенные в объект через переменную y, будут также видны через переменную x.

К наиболее распространенным действиям, выполняемым с объектами, относятся их создание и установка, запрашивание, удаление, проверка и перечисление свойств объектов. Такие фундаментальные операции будут описаны в предстоящих разделах главы. После них будут рассматриваться более сложные темы.

Свойство имеет имя и значение. Именем свойства может быть любая строка, включая пустую строку (или любое значение Symbol), но никакой объект не может иметь два свойства с одним и тем же именем. Значением свойства может быть любое значение JavaScript либо функция получения или установки (или обе). Функции получения и установки обсуждаются в подразделе 6.10.6.

Временами важно иметь возможность проводить различие между свойствами, определяемыми напрямую объектом, и свойствами, которые унаследованы от объекта прототипа. Свойства, которые не были унаследованы, в JavaScript называются *собственными свойствами*.

В дополнение к имени и значению каждое свойство имеет три атрибута свойства:

- атрибут `writable` (допускает запись) указывает, можно ли устанавливать значение свойства;
- атрибут `enumerable` (допускает перечисление) указывает, возвращается ли имя свойства в цикле `for/in`;
- атрибут `configurable` (допускает конфигурирование) указывает, можно ли удалять свойство и изменять его атрибуты.

Многие встроенные объекты JavaScript имеют свойства, которые допускают только чтение, не разрешают перечисление или не поддерживают конфигурирование. Тем не менее, по умолчанию все свойства создаваемых вами объектов допускают запись, перечисление и конфигурирование. В разделе 14.1 объясняются методики, позволяющие указывать нестандартные значения атрибутов свойств для ваших объектов.

6.2. Создание объектов

Объекты можно создавать с помощью объектных литералов, ключевого слова new и функции `Object.create()`. Все приемы рассматриваются в последующих подразделах.

6.2.1. Объектные литералы

Самый легкий способ создания объекта предусматривает включение в код JavaScript объектного литерала. В своей простейшей форме *объектный литерал* представляет собой разделенный запятыми список пар имя:значение, заключенный в фигурные скобки. Имя свойства — это идентификатор JavaScript или строка (допускается и пустая строка). Значение свойства — это любое выражение JavaScript; значение выражения (оно может быть элементарным или объектным значением) становится значением свойства. Ниже приведены примеры:

```
let empty = {};                           // Объект без свойств.
let point = { x: 0, y: 0 };               // Два числовых свойства.
let p2 = { x: point.x, y: point.y+1 };    // Более сложные значения.
let book = {
    "main title": "JavaScript",           // Имена этих свойств включают
                                          // пробелы и дефисы,
    "sub-title": "The Definitive Guide", // а потому для них исполь-
                                          // зуются строковые литералы.
    for: "all audiences",         // for - зарезервированное слово,
                                  // но без кавычек.
    author: {                     // Значением свойства this является
        firstname: "David",       // сам объект.
        surname: "Flanagan"
    }
};
```

За последним свойством в объектном литерале разрешено помещать хвостовую запятую. Некоторые стили программирования поощряют применение хвостовых запятых, так что вы с меньшей вероятностью получите синтаксическую ошибку при добавлении нового свойства в конец объектного литерала когда-нибудь в будущем.

Объектный литерал — это выражение, которое создает и инициализирует новый отдельный объект каждый раз, когда оно вычисляется. Значение каждого свойства вычисляется при каждом вычислении литерала. Таким образом, единственный объектный литерал может создавать много новых объектов, если он находится внутри тела цикла или в функции, которая вызывается неоднократно, и значения свойств создаваемых объектов могут отличаться друг от друга.

В показанных здесь объектных литералах используется простой синтаксис, который был законным, начиная с самых ранних версий JavaScript. В последних версиях языка появилось несколько новых возможностей, касающихся объектных литералов, которые раскрываются в разделе 6.10.

6.2.2. Создание объектов с помощью операции new

Операция new создает и инициализирует новый объект. За ключевым словом new должен следовать вызов функции. Применяемая подобным способом функция называется *конструктором* и предназначена для инициализации вновь созданного объекта. В JavaScript предусмотрены конструкторы для встроенных типов, например:

```
let o = new Object();   // Создает пустой объект: то же, что и {}
let a = new Array();    // Создает пустой массив: то же, что и []
let d = new Date(); // Создает объект Date, представляющий текущее время
let r = new Map();  // Создает объект Map для отображения ключ/значение
```

В дополнение к таким встроенным конструкторам общепринято определять собственные функции конструкторов для инициализации создаваемых новых объектов. Мы обсудим это в главе 9.

6.2.3. Прототипы

Чтобы можно было приступить к исследованию третьей методики создания объектов, необходимо уделить внимание прототипам. Почти с каждым объектом JavaScript ассоциирован второй объект JavaScript, который называется *прототипом*, и первый объект наследует свойства от прототипа.

Все объекты, создаваемые объектными литералами, имеют тот же самый объект-прототип, на который можно ссылаться в коде JavaScript как на `Object.prototype`. Объекты, создаваемые с использованием ключевого слова `new` и вызова конструктора, применяют в качестве своих прототипов значение свойства `prototype` функции конструктора. Таким образом, объект, созданный посредством `new Object()`, наследует `Object.prototype`, как и объект, созданный с помощью `{}`. Аналогично объект, созданный с использованием `new Array()`, применяет в качестве прототипа `Array.prototype`, а объект, созданный с использованием `new Date()`, получает прототип `Date.prototype`. В начале изучения JavaScript такая особенность может сбивать с толку. Запомните: почти все объекты имеют *прототип*, но только относительно небольшое количество объектов располагают свойством `prototype`. Именно эти объекты со свойствами `prototype` определяют *прототипы* для всех остальных объектов.

`Object.prototype` — один из редких объектов, не имеющих прототипов: он не наследует никаких свойств. Другие объекты-прототипы являются нормальными объектами, которые имеют прототип.

Большинство встроенных конструкторов (и большинство определяемых пользователем конструкторов) имеют прототип, унаследованный от `Object.prototype`. Скажем, `Date.prototype` наследует свойства от `Object.prototype`, поэтому объект `Date`, созданный посредством `new Date()`, наследует свойства от `Date.prototype` и `Object.prototype`. Такая связанная последовательность объектов-прототипов называется *цепочкой прототипов*.

Работа наследования свойств объясняется в подразделе 6.3.2. В главе 9 более подробно обсуждается связь между прототипами и конструкторами: в ней показано, как определять новые "классы" объектов за счет написания функции конструктора и установки ее свойства `prototype` в объект-прототип, который должен применяться "экземплярами", созданными с помощью этого конструктора. В разделе 14.3 мы выясним, каким образом запрашивать (и даже изменять) прототип объекта.

6.2.4. Object.create()

Object.create() создает новый объект, используя в качестве его прототипа первый аргумент:

```
let o1 = Object.create({x: 1, y: 2});// o1 наследует свойства x и y.
o1.x + o1.y                          // => 3
```

Вы можете передать null, чтобы создать новый объект, не имеющий прототипа, но в таком случае вновь созданный объект ничего не унаследует, даже базовые методы вроде toString() (т.е. он не будет работать с операцией +):

```
let o2 = Object.create(null);        // o2 не наследует ни свойства,
                                     // ни методы
```

Если вы хотите создать обыкновенный пустой объект (подобный объекту, возвращаемому {} или new Object()), тогда передайте Object.prototype:

```
let o3 = Object.create(Object.prototype); //o3 подобен {} или new Object()
```

Возможность создания нового объекта с помощью произвольного прототипа является мощным инструментом, и мы будем применять Object.create() в нескольких местах данной главы. (Object.create() также принимает необязательный второй аргумент, который описывает свойства нового объекта. Второй аргумент относится к расширенным средствам и раскрывается в разделе 14.1.)

Один из случаев использования Object.create() — когда вы хотите защититься от непреднамеренной (но не злоумышленной) модификации объекта библиотечной функцией, над которой вы не имеете контроля. Вместо передачи объекта напрямую функции вы можете передать объект, унаследованный от него. Если функция читает свойства этого объекта, то она будет видеть унаследованные значения. Однако если она устанавливает свойства, то запись не будет влиять на первоначальный объект.

```
let o = { x: "не изменяйте это значение" };
library.function(Object.create(o)); //Защита от случайной модификации
```

Для понимания, почему это работает, необходимо знать, как свойства запрашиваются и устанавливаются в JavaScript. Мы рассмотрим обе темы в следующем разделе.

6.3. Запрашивание и установка свойств

Чтобы получить значение свойства, применяйте точку (.) или квадратные скобки ([]), описанные в разделе 4.4. В левой стороне должно быть выражение, значением которого является объект. Если используется точка, тогда с правой стороны должен находиться простой идентификатор, именующий свойство. В случае применения квадратных скобок значение внутри скобок должно быть выражением, вычисляемым в строку, которая содержит имя желаемого свойства:

```
let author = book.author;  //Получить свойство "author" объекта book
let name = author.surname; //Получить свойство "surname" объекта author
let title = book["main title"]; // Получить свойство "main title"
                                // объекта book
```

Для создания набора свойств используйте точку или квадратные скобки, как делалось при запрашивании свойства, но в левой стороне выражения присваивания:

```
book.edition = 7;          // Создать свойство "edition" объекта book
book["main title"] = "ECMAScript";  // Изменить свойство "main title"
```

Когда применяется запись с квадратными скобками, мы говорим, что выражение внутри квадратных скобок должно вычисляться в строку или, формулируя более точно, выражение должно вычисляться в строку или значение, которое может быть преобразовано в строку или значение `Symbol` (см. подраздел 6.10.3). Например, в главе 7 мы увидим, что внутри квадратных скобок принято использовать числа.

6.3.1. Объекты как ассоциативные массивы

Как объяснялось в предыдущем разделе, два показанных ниже выражения JavaScript имеют одно и то же значение:

```
object.property
object["property"]
```

Первый синтаксис с точкой и идентификатором похож на синтаксис, применяемый для доступа к статическому полю структуры или объекта в C или Java. Второй синтаксис с квадратными скобками и строкой выглядит как доступ в массив, но индексированный не по числам, а по строкам. Такой вид массива известен как *ассоциативный массив* (а также хеш, отображение или словарь). Объекты JavaScript являются ассоциативными массивами, и в этом разделе объясняется, почему они важны.

В C, C++, Java и похожих строго типизированных языках объект может иметь только фиксированное количество свойств, а имена свойств должны быть определены заранее. Поскольку JavaScript — слабо типизированный язык, такое правило не применяется: программа способна создавать любое количество свойств в любом объекте. Тем не менее, при использовании операции `.` для доступа к свойству объекта имя свойства выражается как идентификатор. Идентификаторы в программе JavaScript должны набираться буквально; они не относятся к типам данных, а потому ими нельзя манипулировать в программе.

С другой стороны, когда вы обращаетесь к свойству объекта с помощью записи в форме массива `[]`, имя свойства выражается в виде строки. Строки являются типом данных JavaScript, так что их можно создавать и манипулировать ими во время выполнения программы. Таким образом, например, следующий код JavaScript будет допустимым:

```
let addr = "";
for(let i = 0; i < 4; i++) {
   addr += customer[`address${i}`] + "\n";
}
```

Код читает и производит конкатенацию свойств `address0`, `address1`, `address2` и `address3` объекта `customer`.

Приведенный короткий пример демонстрирует гибкость, которую обеспечивает применение записи в форме массива для доступа к свойствам объекта со строковыми выражениями. Код можно было бы переписать с использованием точечной записи, но бывают ситуации, когда подходит только запись в форме массива. Предположим, например, что вы пишете программу, где задействуете сетевые ресурсы для расчета текущей стоимости акций пользователя на фондовой бирже. Программа позволяет пользователю вводить название компании, акциями которой он владеет, и количество акций каждой компании. Для хранения такой информации вы можете применять объект по имени `portfolio`. Объект имеет по одному свойству на акции каждой компании. Именем свойства будет название компании, а значением свойства — количество акций данной компании. Таким образом, если пользователь владеет 50 акциями компании IBM, тогда свойство `portfolio.ibm` имеет значение 50.

Частью этой программы может быть функция для добавления нового пакета акций в портфель ценных бумаг:

```
function addstock(portfolio, stockname, shares) {
   portfolio[stockname] = shares;
}
```

Так как пользователь вводит названия компаний по время выполнения, узнать заранее имена свойств не удастся. Поскольку вы не знаете имена свойств при написании программы, то не можете использовать операцию . для доступа к свойствам объекта `portfolio`. Однако вы можете применять операцию `[]`, потому что для именования свойства в ней используется строковое значение (которое динамическое и может изменяться во время выполнения), а не идентификатор (который статический и должен быть жестко закодированным в программе).

В главе 5 был представлен цикл `for/in` (и в разделе 6.6 мы снова к нему вернемся). Мощь этого оператора JavaScript становится ясной, когда подумать о его применении с ассоциативными массивами. Вот как можно было бы использовать цикл `for/in` при подсчете общей стоимости портфеля ценных бумаг:

```
function computeValue(portfolio) {
   let total = 0.0;
   for(let stock in portfolio) {     // Для каждого пакета акций
                                     // в портфеле ценных бумаг:
      let shares = portfolio[stock]; // получить количество акций
      let price = getQuote(stock);   // найти курс акций
      total += shares * price;       // добавить стоимость пакета
                                     // акций к общей стоимости
   }
   return total;                     // Возвратить общую стоимость
}
```

С объектами JavaScript обычно взаимодействуют как с ассоциативными массивами и важно понимать особенности работы с ними. Тем не менее, в ES6 и последующих версиях класс `Map`, рассматриваемый в подразделе 11.1.2, часто оказывается более удачным вариантом, нежели создание простого объекта.

6.3.2. Наследование

Объекты JavaScript имеют набор "собственных свойств" и вдобавок наследуют набор свойств от своих объектов-прототипов. Чтобы разобраться в наследовании, мы должны обсудить доступ к свойствам более подробно. В примерах, приводимых в этом разделе, будет применяться функция `Object.create()` для создания объектов с указанными прототипами. Однако в главе 9 мы увидим, что каждый раз, когда с помощью `new` создается экземпляр класса, на самом деле создается объект, который наследует свойства от объекта-прототипа.

Предположим, что вы запрашиваете свойство x в объекте o. Если в o отсутствует собственное свойство с таким именем, тогда свойство x запрашивается в объекте-прототипе o[1]. Если объект-прототип не имеет собственного свойства по имени x, но сам располагает прототипом, то запрос выполняется для прототипа объекта-прототипа. Процесс продолжается до тех пор, пока свойство x не будет найдено или не обнаружится объект с прототипом `null`. Как должно быть понятно, атрибут `prototype` объекта создает цепочку или связный список, от которого наследуются свойства:

```
let o = {};     // o наследует методы объекта от Object.prototype
o.x = 1;        // и теперь имеет собственное свойство x.
let p = Object.create(o); //p наследует свойства от o и Object.prototype
p.y = 2;        // и имеет собственное свойство y.
let q = Object.create(p); //q наследует свойства от p, o и...
q.z = 3;        // ...Object.prototype и имеет собственное свойство z.
let f = q.toString();     //toString наследуется от Object.prototype
q.x + q.y       // => 3; x и y наследуются от o и p
```

Теперь предположим, что вы присваиваете значение свойству x объекта o. Если o уже имеет собственное (не унаследованное) свойство по имени x, тогда присваивание просто изменяет значение этого существующего свойства. В противном случае присваивание создает в объекте o новое свойство по имени x. Если объект o ранее унаследовал свойство x, то унаследованное свойство скрывается вновь созданным свойством с тем же самым именем.

Присваивание свойства просматривает цепочку прототипов лишь для определения, разрешено ли присваивание. Скажем, если o наследует свойство по имени x, допускающее только чтение, тогда присваивание не разрешается. (Детали о том, когда свойство может устанавливаться, ищите в подразделе 6.3.3.) Тем не менее, если присваивание разрешено, то оно всегда создает или устанавливает свойство в первоначальном объекте и никогда не модифицирует объекты в цепочке прототипов. Тот факт, что наследование действует при запрашивании свойств, но не при их установке, является ключевой особенностью языка JavaScript, т.к. появляется возможность выборочно переопределять унаследованные свойства:

[1] Вспомните, что почти все объекты располагают прототипом, но большинство не имеет свойства по имени `prototype`. Наследование JavaScript работает, даже если вы не можете получить доступ к объекту-прототипу напрямую. Но если вы хотите узнать, как это делать, тогда обратитесь в раздел 14.3.

```
let unitcircle = { r: 1 };           // Объект, от которого будет
                                     // делаться наследование
let c = Object.create(unitcircle);   // c наследует свойство r
c.x = 1; c.y = 1;   // c определяет два собственных свойства
c.r = 2;            // c переопределяет свое унаследованное свойство
unitcircle.r        // => 1: прототип не затронут
```

Существует одно исключение из правила о том, что присваивание свойства либо терпит неудачу, либо создает или устанавливает свойство в первоначальном объекте. Если o наследует свойство x, и оно является свойством средства доступа с методом установки (см. подраздел 6.10.6), тогда вместо создания нового свойства x в o вызывается данный метод установки. Однако имейте в виду, что метод установки вызывается с объектом o, а не объектом-прототипом, который определяет свойство, поэтому если метод установки определяет любые свойства, то они попадают в объект o, снова оставляя цепочку прототипов незатронутой.

6.3.3. Ошибки доступа к свойствам

Выражения доступа к свойствам не всегда возвращают или устанавливают значение. В настоящем разделе будет показано, что может пойти не так при запрашивании или установке свойства. Запрашивание свойства, которое не существует, не считается ошибкой. Если свойство x не найдено как собственное или унаследованное свойство объекта o, то вычисление выражения доступа к свойству `o.x` дает значение `undefined`. Вспомните, что наш объект `book` имеет свойство `"sub-title"`, но не `"subtitle"`:

```
book.subtitle                        // => undefined: свойство не существует
```

Тем не менее, попытка запрашивания свойства объекта, который не существует, является ошибкой. Значения `null` и `undefined` не имеют свойств, и запрашивание свойств таких значений будет ошибкой. Продолжим предыдущий пример:

```
let len = book.subtitle.length; // !TypeError: undefined не имеет
                                // свойства length
```

Выражения доступа к свойствам терпят неудачу, если с левой стороны операции `.` находится `null` или `undefined`. Таким образом, при написании выражения вроде `book.author.surname` нужно соблюдать осторожность, когда вы не уверены в том, что `book` и `book.author` действительно определены. Защититься от возникновения проблемы такого рода можно двумя способами:

```
// Многословная и явная методика
let surname = undefined;
if (book) {
    if (book.author) {
        surname = book.author.surname;
    }
}
// Лаконичная и идиоматическая альтернатива
// для получения surname либо null или undefined
surname = book && book.author && book.author.surname;
```

Чтобы понять, почему такое идиоматическое выражение позволяет предотвратить генерацию исключений TypeError, у вас может возникнуть необходимость еще раз почитать о поведении с коротким замыканием операции && в подразделе 4.10.1.

Как было описано в подразделе 4.4.1, в ES2020 поддерживается условный доступ к свойствам с помощью ?., что позволяет переписать предыдущее выражение присваивания следующим образом:

```
let surname = book?.author?.surname;
```

Попытка установки свойства для null или undefined также генерирует TypeError. Попытки установки свойств для остальных значений тоже не всегда оказываются успешными: некоторые свойства допускают только чтение и не могут быть установлены, а некоторые объекты не разрешают добавление новых свойств. В строгом режиме (см. подраздел 5.6.3) исключение TypeError генерируется всякий раз, когда попытка установки свойства терпит неудачу. За рамками строгого режима такие неудачи обычно проходят молча.

Правила, которые определяют, когда присваивание свойства проходит успешно, а когда терпит неудачу, интуитивно понятны, но их трудно выразить в лаконичной форме. Попытка установки свойства p объекта o терпит неудачу в следующих обстоятельствах.

- o имеет собственное свойство p, допускающее только чтение: устанавливать свойства только для чтения невозможно.
- o имеет унаследованное свойство p, допускающее только чтение: скрывать унаследованное свойство только для чтения собственным свойством с таким же именем невозможно.
- o не имеет собственного свойства p; o не наследует свойство p с методом установки и атрибут extensible (см. раздел 14.2) объекта o равен false. Поскольку p не существует в o и метод установки отсутствует, то свойство p должно быть добавлено к o. Но если объект o не допускает расширения, тогда в нем нельзя определять новые свойства.

6.4. Удаление свойств

Операция delete (см. подраздел 4.13.4) удаляет свойство из объекта. Ее единственным операндом должно быть выражение доступа к свойству. Удивительно, но delete работает не со значением свойства, а с самим свойством:

```
delete book.author;     // Объект book теперь не имеет свойства author
delete book["main title"]; // А теперь он не имеет и свойства "main title"
```

Операция delete удаляет только собственные свойства, но не унаследованные. (Чтобы удалить унаследованное свойство, нужно удалять его из объекта-прототипа, где оно определено. Такое действие повлияет на каждый объект, унаследованный от этого прототипа.)

Выражение delete вычисляется как true, если удаление прошло успешно или было безрезультатным (наподобие удаления несуществующего свойства).

Кроме того, delete вычисляется как true, когда применяется (бессмысленно) с выражением, не являющимся выражением доступа к свойству:

```
let o = {x: 1};     // o имеет собственное свойство x и наследует
                    // свойство toString
delete o.x          // => true: свойство x удаляется
delete o.x          // => true: ничего не делает
                    //          (x не существует), но в любом случае true
delete o.toString   // => true: ничего не делает
                    //          (toString - не собственное свойство)
delete 1            // => true: бессмысленно, но в любом случае true
```

Операция delete не удаляет свойства, которые имеют атрибут configurable, установленный в false. Некоторые свойства встроенных объектов не поддерживают конфигурирование, т.к. представляют собой свойства глобального объекта, созданного объявлениями переменных и функций. В строгом режиме попытка удаления свойства, не допускающего конфигурирование, приводит к генерации TypeError. В нестрогом режиме delete в таком случае просто вычисляется как false:

```
// В строгом режиме все показанные ниже удаления генерируют TypeError,
// а не возвращают false
delete Object.prototype //=>false: свойство не допускает конфигурирование
var x = 1;              // Объявление глобальной переменной
delete globalThis.x     // => false: это свойство нельзя удалить
function f() {}         // Объявление глобальной функции
delete globalThis.f     // => false: это свойство тоже нельзя удалить
```

При удалении конфигурируемых свойств глобального объекта в нестрогом режиме вы можете опускать ссылку на глобальный объект и указывать после операции delete имя свойства:

```
globalThis.x = 1;       // Создать конфигурируемое свойство
                        // глобального объекта (никаких let или var)
delete x                // => true: это свойство можно удалить
```

Однако в строгом режиме операция delete генерирует SyntaxError, если ее операндом оказывается неуточненный идентификатор вроде x, и вы должны обеспечить явный доступ к свойству:

```
delete x;               // SyntaxError в строгом режиме
delete globalThis.x;    // Это работает
```

6.5. Проверка свойств

Объекты JavaScript можно рассматривать как наборы свойств и зачастую полезно иметь возможность проверять членство свойства в таком наборе — с целью выяснения, имеет ли объект свойство с заданным именем. Это делается с помощью операции in, посредством методов hasOwnProperty() и propertyIsEnumerable() или просто путем запрашивания свойства. Во всех приведенных здесь примерах в качестве имен свойств используются строки, но они также работают со значениями Symbol (см. подраздел 6.10.3).

Операция in ожидает с левой стороны имя свойства и с правой стороны объект. Она возвращает true, если объект имеет собственное или унаследованное свойство с указанным именем:

```
let o = { x: 1 };
"x" in o           // => true: o имеет собственное свойство "x"
"y" in o           // => false: o не имеет свойства "y"
"toString" in o    // => true: o наследует свойство toString
```

Метод The hasOwnProperty() объекта проверяет, имеет ли данный объект собственное свойство с заданным именем. Для унаследованных свойств он возвращает false:

```
let o = { x: 1 };
o.hasOwnProperty("x")       // => true: o имеет собственное свойство x
o.hasOwnProperty("y")       // => false: o не имеет свойства y
o.hasOwnProperty("toString") // => false: toString - унаследованное
                            //           свойство
```

Метод propertyIsEnumerable() улучшает проверку hasOwnProperty(). Он возвращает true, только если именованное свойство является собственным и атрибут enumerable имеет значение true. Определенные встроенные свойства не поддерживают перечисление. Свойства, созданные нормальным кодом JavaScript, перечислимы при условии, что вы не применяли одну из методик, показанных в разделе 14.1, чтобы сделать их не перечислимыми.

```
let o = { x: 1 };
o.propertyIsEnumerable("x") // => true: o имеет собственное
                            //          перечислимое свойство x
o.propertyIsEnumerable("toString")  //=>false:не собственное свойство
Object.prototype.propertyIsEnumerable("toString")  // => false:
                                                   // не перечислимое свойство
```

Вместо использования операции in часто достаточно просто запросить свойство и применить !== для его проверки на предмет undefined:

```
let o = { x: 1 };
o.x !== undefined           // => true: o имеет свойство x
o.y !== undefined           // => false: o не имеет свойства y
o.toString !== undefined    // => true: o наследует свойство toString
```

Операция in может делать то, на что не способна показанная здесь простая методика доступа к свойствам. Операция in проводит различие между свойствами, которые не существуют, и свойствами, которые существуют, но были установлены в undefined. Взгляните на такой код:

```
let o = { x: undefined }; // Свойство явно устанавливается в undefined
o.x !== undefined         // => false: свойство существует,
                          //           но равно undefined
o.y !== undefined         // => false: свойство не существует
"x" in o                  // => true: свойство существует
"y" in o                  // => false: свойство не существует
delete o.x;               // Удаление свойства x
"x" in o                  // => false: свойство больше не существует
```

6.6. Перечисление свойств

Вместо проверки индивидуальных свойств на предмет существования временами мы хотим пройти по ним или получить список всех свойств объекта. Решить задачу можно несколькими способами.

Цикл `for/in` был раскрыт в подразделе 5.4.5. Он выполняет тело цикла по одному разу для каждого перечислимого свойства (собственного или унаследованного) указанного объекта, присваивая имя свойства переменной цикла. Встроенные методы, наследуемые объектами, не являются перечислимыми, но свойства, которые ваш код добавляет к объектам, по умолчанию будут перечислимыми. Например:

```
let o = {x: 1, y: 2, z: 3}; // Три перечислимых собственных свойства
o.propertyIsEnumerable("toString") // => false: не перечислимое
for(let p in o) {           // Проход в цикле по свойствам
   console.log(p);          // Выводится x, y и z, но не toString
}
```

Чтобы избежать перечисления унаследованных свойств посредством `for/in`, вы можете поместить внутрь тела цикла явную проверку:

```
for(let p in o) {
   if (!o.hasOwnProperty(p)) continue;  // Пропускать унаследованные
                                        // свойства
}
for(let p in o) {
   if (typeof o[p] === "function") continue;  // Пропускать все методы
}
```

В качестве альтернативы использованию цикла `for/in` часто легче получить массив имен свойств для объекта и затем проходить по этому массиву в цикле `for/of`. Есть четыре функции, которые можно применять для получения массива имен свойств.

- Функция `Object.keys()` возвращает массив имен перечислимых собственных свойств объекта. Она не включает не перечислимые свойства, унаследованные свойства или свойства с именами, представленными посредством значений `Symbol` (см. подраздел 6.10.3).
- Функция `Object.getOwnPropertyNames()` работает подобно `Object.keys()`, но возвращает массив также имен не перечислимых собственных свойств при условии, что их имена представлены строками.
- Функция `Object.getOwnPropertySymbols()` возвращает собственные свойства, имена которых являются значениями `Symbol`, перечислимые они или нет.
- Функция `Reflect.ownKeys()` возвращает имена всех собственных свойств, перечислимых и не перечислимых, представленных как строками, так и значениями `Symbol`. (См. раздел 14.6.)

Примеры использования `Object.keys()` с циклом `for/of` приведены в разделе 6.7.

6.6.1. Порядок перечисления свойств

В ES6 формально определен порядок, в котором перечисляются собственные свойства объекта. `Object.keys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()`, `Reflect.ownKeys()` и связанные методы, такие как `JSON.stringify()`, перечисляют свойства в следующем порядке с учетом своих дополнительных ограничений на то, перечисляют ли они не перечислимые свойства или свойства, чьи имена представлены строками или значениями `Symbol`.

- Строковые свойства, имена которых являются неотрицательными целыми числами, перечисляются первыми в числовом порядке от меньших к большим. Это правило означает, что свойства массивов и подобных им объектов будут перечисляться по порядку.
- После того, как перечислены все свойства, которые выглядят похожими на индексы массива, перечисляются все оставшиеся свойства со строковыми именами (включая свойства, которые выглядят как отрицательные числа или числа с плавающей точкой). Такие свойства перечисляются в порядке, в котором они добавлялись к объекту. Для свойств, определенных в объектном литерале, этим порядком будет порядок их следования в литерале.
- Наконец, свойства с именами, представленными с помощью значений `Symbol`, перечисляются в порядке, в котором они добавлялись к объекту.

Порядок перечисления для цикла `for/in` определен не настолько строго, как для упомянутых выше функций перечисления, но разные реализации перечисляют собственные свойства в только что описанном порядке, после чего перемещаются вверх по цепочке прототипов, перечисляя свойства в том же порядке для каждого объекта-прототипа. Тем не менее, обратите внимание, что свойство не будет перечисляться, если свойство с тем же самым именем уже подвергалось перечислению или даже когда уже просматривалось не перечислимое свойство с таким же именем.

6.7. Расширение объектов

Необходимость копирования свойств одного объекта в другой — обычная операция в программах JavaScript. Ее легко реализовать с помощью такого кода:

```
let target = {x: 1}, source = {y: 2, z: 3};
for(let key of Object.keys(source)) {
   target[key] = source[key];
}
target   // => {x: 1, y: 2, z: 3}
```

Но из-за того, что операция копирования настолько распространена, в различных фреймворках JavaScript для ее выполнения были определены служебные функции, часто имеющие имя `extend()`. Наконец, в версии ES6 эта возможность стала частью базового языка JavaScript в форме `Object.assign()`.

Функция `Object.assign()` ожидает получения в своих аргументах двух и более объектов. Она модифицирует и возвращает первый аргумент, в котором указан целевой объект, но не изменяет второй и любой последующий аргумент, где указаны исходные объекты. Для каждого исходного объекта функция `Object.assign()` копирует его перечислимые собственные свойства (включая те, имена которых являются значениями `Symbol`) в целевой объект. Она обрабатывает исходные объекты в порядке их следования в списке аргументов, так что свойства первого исходного объекта переопределяют свойства с такими же именами в целевом объекте, а свойства второго исходного объекта (если он указан) переопределяют свойства с такими же именами в первом исходном объекте.

Функция `Object.assign()` копирует свойства с помощью обычных операций получения и установки свойств, поэтому если исходный объект располагает методом получения или целевой объект имеет метод установки, то они будут вызваны во время копирования, но сами не скопируются.

Одна из причин передачи свойств из одного объекта другому — когда имеется объект, который определяет стандартные значения для множества свойств, и необходимо скопировать такие стандартные свойства в другой объект, если свойства с теми же самыми именами в нем пока не существуют. Наивное применение `Object.assign()` не обеспечит то, что нужно:

```
Object.assign(o, defaults);     // Переопределяет все в
                                // o стандартными свойствами
```

Взамен вы можете создать новый объект, скопировать в него стандартные свойства и затем переопределить их посредством свойств в `o`:

```
o = Object.assign({}, defaults, o);
```

В подразделе 6.10.4 будет показано, что выразить такое действие копирования и переопределения можно также с использованием операции распространения `...`, например:

```
o = {...defaults, ...o};
```

Мы могли бы избежать накладных расходов, связанных с созданием и копированием дополнительного объекта, написав версию функции `Object.assign()`, которая копирует свойства, только если они отсутствуют:

```
// Похожа на Object.assign(), но не переопределяет существующие
// свойства (и также не обрабатывает свойства Symbol).
function merge(target, ...sources) {
    for(let source of sources) {
        for(let key of Object.keys(source)) {
            if (!(key in target)) { // Это отличается от Object.assign()
                target[key] = source[key];
            }
        }
    }
    return target;
}
Object.assign({x: 1},{x: 2, y: 2}, {y: 3, z: 4}) // => {x: 2, y: 3, z: 4}
merge({x: 1}, {x: 2, y: 2}, {y: 3, z: 4})        // => {x: 1, y: 2, z: 4}
```

Довольно легко реализовать и другие утилиты манипулирования свойствами, подобные функции `merge()`. Скажем, функция `restrict()` могла бы удалять свойства объекта, если они отсутствуют в другом, шаблонном объекте. Или же функция `subtract()` могла бы удалять все свойства, имеющиеся в одном объекте, из другого объекта.

6.8. Сериализация объектов

Сериализация объектов представляет собой процесс преобразования состояния объекта в строку, из которой позже он может быть восстановлен. Функции `JSON.stringify()` и `JSON.parse()` сериализируют и восстанавливают объекты JavaScript. Эти функции работают с форматом обмена данными JSON, который означает "JavaScript Object Notation" (представление объектов JavaScript) и имеет синтаксис, очень похожий на синтаксис литералов объектов и массивов JavaScript:

```
let o = {x: 1, y: {z: [false, null, ""]}};    // Определение тестового
                                              // объекта
let s = JSON.stringify(o); //s == '{"x":1,"y":{"z":[false,null,""]}}'
let p = JSON.parse(s);     // p == {x: 1, y: {z: [false, null, ""]}}
```

Синтаксис JSON является *подмножеством* синтаксиса JavaScript и не способен представлять все значения JavaScript. Поддерживается возможность сериализации и восстановления объектов, массивов, строк, конечных чисел, `true`, `false` и `null`. Значения `NaN`, `Infinity` и `-Infinity` сериализируются в `null`. Объекты `Date` сериализируются в строки дат формата ISO (см. функцию `Date.toJSON()`), но `JSON.parse()` оставляет их в строковой форме и не восстанавливает первоначальные объекты `Date`. Объекты `Function`, `RegExp` и `Error` и значение `undefined` не могут сериализироваться или восстанавливаться. `JSON.stringify()` сериализирует только перечислимые собственные свойства объекта. Если значение свойства не может быть сериализировано, тогда такое свойство просто опускается из строкового вывода. Функции `JSON.stringify()` и `JSON.parse()` принимают необязательный второй аргумент, который можно применять для настройки процесса сериализации и/или восстановления, указывая список свойств, подлежащих сериализации, например, или преобразуя определенные значения во время процесса сериализации либо превращения в строки. Полное описание упомянутых функций приведено в разделе 11.6.

6.9. Методы `Object`

Как обсуждалось ранее, все объекты JavaScript (кроме явно созданных без прототипа) наследуют свойства от `Object.prototype`. К таким свойствам в основном относятся методы, которые из-за своей повсеместной доступности представляют особый интерес для программистов на JavaScript. Скажем, мы уже встречались с методами `hasOwnProperty()` и `propertyIsEnumerable()`. (Вдобавок мы уже раскрыли немало статических функций, определяемых в

конструкторе Object, например, Object.create() и Object.keys().) В текущем разделе объясняется несколько универсальных объектных методов, которые определены в Object.prototype, но предназначены для замены другими, более специализированными реализациями. В последующих подразделах будут показаны примеры определения этих методов в одиночном объекте. В главе 9 вы узнаете, как определять такие методы более универсально для целого класса объектов.

6.9.1. Метод toString()

Метод toString() не имеет аргументов; он возвращает строку, каким-то образом представляющую значение объекта, на котором он вызван. Интерпретатор JavaScript вызывает данный метод всякий раз, когда объект необходимо преобразовать в строку. Подобное происходит, например, при использовании операции + для конкатенации строки с объектом или при передаче объекта методу, который ожидает строку.

Стандартный метод toString() не особо информативен (хотя он полезен для выяснения класса объекта, как будет показано в подразделе 14.4.3). Скажем, следующий код дает в результате строку "[object Object]":

```
let s = { x: 1, y: 1 }.toString();    // s == "[object Object]"
```

Поскольку стандартный метод toString() не предоставляет достаточный объем полезной информации, многие классы определяют собственные версии toString(). Например, когда массив преобразуется в строку, вы получаете список элементов массива, каждый из которых сам преобразован в строку, а когда функция преобразуется в строку, вы получаете исходный код для функции. Вот как вы могли бы определить собственный метод toString():

```
let point = {
    x: 1,
    y: 2,
    toString: function() { return `(${this.x}, ${this.y})`; }
};
String(point)                    // => "(1, 2)": toString() применяется
                                 // для преобразований в строки
```

6.9.2. Метод toLocaleString()

В дополнение к базовому методу toString() все объекты имеют метод toLocaleString(), который предназначен для возвращения локализованного строкового представления объекта. Стандартный метод toLocaleString(), определенный в Object, никакой локализации не делает: он просто вызывает метод toString() и возвращает его значение. В классах Date и Number определены настроенные версии toLocaleString(), которые пытаются форматировать числа, даты и время в соответствии с локальными соглашениями. В классе Array определен метод toLocaleString(), который работает подобно toString(), но форматирует элементы массива, вызывая их методы toLocaleString(), а не toString(). Вы можете делать то же самое с объектом Point:

```
let point = {
   x: 1000,
   y: 2000,
   toString: function() { return `(${this.x}, ${this.y})`; },
   toLocaleString: function() {
      return `(${this.x.toLocaleString()}, ${this.y.toLocaleString()})`;
   }
};
point.toString()           // => "(1000, 2000)"
point.toLocaleString()     // => "(1,000, 2,000)": обратите внимание
                           // на наличие разделителей тысяч
```

При реализации метода `toLocaleString()` могут оказаться полезными классы интернационализации, описанные в разделе 11.7.

6.9.3. Метод `valueOf()`

Метод `valueOf()` во многом похож на метод `toString()`, но вызывается, когда интерпретатору JavaScript необходимо преобразовать объект в какой-то элементарный тип, отличающийся от строки — обычно в число. Интерпретатор JavaScript вызывает `valueOf()` автоматически, если объект используется в контексте, где требуется элементарное значение. Стандартный метод `valueOf()` не делает ничего интересного, но некоторые встроенные классы определяют собственные методы `valueOf()`. Класс `Date` определяет метод `valueOf()` для преобразования дат в числа, что позволяет сравнивать объекты `Date` хронологически посредством операций < и >. Вы могли бы делать что-то подобное с объектами `Point`, определив метод `valueOf()`, который возвращает расстояние точки до начала координат:

```
let point = {
   x: 3,
   y: 4,
   valueOf: function() { return Math.hypot(this.x, this.y); }
};
Number(point) // => 5: valueOf() применяется для преобразования в числа
point > 4     // => true
point > 5     // => false
point < 6     // => true
```

6.9.4. Метод `toJSON()`

В `Object.prototype` на самом деле не определен метод `toJSON()`, но метод `JSON.stringify()` (см. раздел 6.8) ищет метод `toJSON()` в каждом объекте, который нужно сериализировать. Если этот метод существует в объекте, подлежащем сериализации, тогда он вызывается и сериализируется его возвращаемое значение, а не первоначальный объект. Класс `Date` (см. раздел 11.4) определяет метод `toJSON()`, который возвращает допускающее сериализацию строковое представление даты. Вы можете делать то же самое с объектом `Point`:

```
let point = {
    x: 1,
    y: 2,
    toString: function() { return `(${this.x}, ${this.y})`; },
    toJSON: function() { return this.toString(); }
};
JSON.stringify([point])    // => '["(1, 2)"]'
```

6.10. Расширенный синтаксис объектных литералов

В недавних версиях JavaScript синтаксис объектных литералов был расширен несколькими полезными путями. Расширения объясняются в последующих подразделах.

6.10.1. Сокращенная запись свойств

Предположим, что у вас есть значения, хранящиеся в переменных x и y, и вы хотите создать объект со свойствами, именованными как x и y, которые будут хранить эти значения. С использованием базового синтаксиса объектных литералов вам пришлось бы дважды повторять каждый идентификатор:

```
let x = 1, y = 2;
let o = {
    x: x,
    y: y
};
```

В ES6 и последующих версиях вы можете отбросить двоеточие и одну копию идентификатора, получив гораздо более простой код:

```
let x = 1, y = 2;
let o = { x, y };
o.x + o.y          // => 3
```

6.10.2. Вычисляемые имена свойств

Иногда вам необходимо создать объект со специфическим свойством, но имя свойства не является константой стадии компиляции, которую вы могли бы буквально набрать в исходном коде. Взамен имя нужного свойства хранится в переменной или возвращается вызовом какой-то функции. Для свойства такого вида применить базовый синтаксис объектных литералов не удастся. Вместо этого вам придется создать объект и затем добавить к нему желаемые свойства как дополнительный шаг:

```
const PROPERTY_NAME = "p1";
function computePropertyName() { return "p" + 2; }
let o = {};
o[PROPERTY_NAME] = 1;
o[computePropertyName()] = 2;
```

Настроить объект подобного рода намного проще с помощью средства ES6, называемого *вычисляемыми свойствами*, которое позволяет переместить квадратные скобки из предыдущего кода прямо внутрь объектного литерала:

```
const PROPERTY_NAME = "p1";
function computePropertyName() { return "p" + 2; }
let p = {
    [PROPERTY_NAME]: 1,
    [computePropertyName()]: 2
};
p.p1 + p.p2    // => 3
```

Квадратные скобки в новом синтаксисе определяют границы произвольного выражения JavaScript, которое вычисляется и результирующее значение (при необходимости преобразованное в строку) используется в качестве имени свойства.

Одна из ситуаций, когда могут понадобиться вычисляемые свойства, возникает при работе с библиотекой кода JavaScript, ожидающей предоставления объектов с заданным набором свойств, имена которых определены как константы в этой библиотеке. Когда вы пишете код для создания объектов, которые будут передаваться библиотеке, то могли бы жестко закодировать имена свойств, но тогда есть риск где-нибудь неправильно набрать имя свойства или столкнуться с проблемами несовместимости версий, если в новой версии библиотеки имена обязательных свойств изменятся. Взамен вы можете обнаружить, что применение синтаксиса вычисляемых свойств с константами имен свойств, определенными в библиотеке, сделало ваш код более надежным.

6.10.3. Символы в качестве имен свойств

Синтаксис вычисляемых свойств открывает доступ к еще одной очень важной возможности объектных литералов. В ES6 и последующих версиях имена свойств могут быть строками или символами. Если вы присвоите символ переменной или константе, то сможете использовать этот символ как имя свойства с применением синтаксиса вычисляемых свойств:

```
const extension = Symbol("my extension symbol");
let o = {
[extension]: { /* в этом объекте хранятся данные расширения */ }
};
o[extension].x = 0;     // Это не будет конфликтовать
                        // с остальными свойствами o
```

Как объяснялось в разделе 3.6, символы являются значениями, трудными для понимания. Вы ничего не можете с ними делать, кроме как использовать их в качестве имен свойств. Однако каждое значение `Symbol` отличается от любого другого значения `Symbol`, т.е. символы хороши для создания уникальных имен свойств. Новое значение `Symbol` создается вызовом фабричной функции `Symbol()`. (Символы представляют собой элементарные значения, не объекты, а потому `Symbol()` — не функция конструктора, которая вызывается

с new.) Значение, возвращаемое Symbol(), не равно никакому другому значению Symbol или значению другого типа. Функции Symbol() можно передавать строку, которая применяется при преобразовании значения Symbol в строку. Но это только средство отладки: два значения Symbol, создаваемые с одним и тем же строковым аргументом, по-прежнему отличаются друг от друга.

Смысл значений Symbol связан не с защитой, а с определением безопасного механизма расширения для объектов JavaScript. Если вы получаете объект из стороннего кода, находящегося вне вашего контроля, и хотите добавить в объект собственные свойства, но гарантировать, что они не будут конфликтовать с любыми существующими свойствами, тогда можете безопасно использовать для имен своих свойств значения Symbol. Поступая подобным образом, вы также можете быть уверены в том, что сторонний код не изменит случайно ваши свойства с символьными именами. (Разумеется, сторонний код мог бы применить Object.getOwnPropertySymbols() для выяснения используемых вами значений Symbol и затем изменить или удалить ваши свойства. Вот почему значения Symbol не являются механизмом защиты.)

6.10.4. Операция распространения

В ES2018 и последующих версиях можно копировать свойства существующего объекта в новый объект, используя внутри объектного литерала "операцию распространения" ...:

```
let position = { x: 0, y: 0 };
let dimensions = { width: 100, height: 75 };
let rect = { ...position, ...dimensions };
rect.x + rect.y + rect.width + rect.height    // => 175
```

В приведенном коде свойства объектов position и dimensions "распространяются" в объектный литерал rect, как если бы они были записаны буквально внутри фигурных скобок. Обратите внимание, что такой синтаксис ... часто называется операцией распространения, но он не является подлинной операцией JavaScript в каком-либо смысле. Это синтаксис особого случая, доступный только внутри объектных литералов. (В других контекстах JavaScript многоточие применяется для других целей, но объектные литералы — единственный контекст, где многоточие вызывает такую вставку одного объекта в другой объект.)

Если и распространяемый объект, и объект, куда он распространяется, имеют свойство с тем же самым именем, тогда значением такого свойства будет то, которое поступит последним:

```
let o = { x: 1 };
let p = { x: 0, ...o };
p.x    // => 1: значение из объекта o переопределяет начальное значение
let q = { ...o, x: 2 };
q.x    // => 2: значение 2 переопределяет предыдущее значение из o
```

Также обратите внимание, что операция распространения распространяет только собственные свойства объекта, но не унаследованные:

```
let o = Object.create({x: 1});  // o наследует свойство x
let p = { ...o };
p.x                             // => undefined
```

В заключение полезно отметить, что хотя операция распространения представлена всего лишь тремя точками в коде, она способна загрузить интерпретатор JavaScript значительным объемом работы. Если объект имеет n свойств, то процесс их распространения в другой объект, вероятно, окажется операцией $O(n)$. Таким образом, когда вы используете `...` внутри цикла или рекурсивной функции в качестве способа сбора данных в одном крупном объекте, то можете написать неэффективный алгоритм $O(n^2)$, который будет плохо масштабироваться с увеличением n.

6.10.5. Сокращенная запись методов

Когда функция определяется как свойство объекта, мы называем ее *методом* (о методах пойдет речь в главах 8 и 9). До версии ES6 вы определяли бы метод в объектном литерале с применением выражения определения функции, как поступали бы в случае любого другого свойства объекта:

```
let square = {
   area: function() { return this.side * this.side; },
   side: 10
};
square.area()    // => 100
```

Тем не менее, в ES6 синтаксис объектных литералов (и также синтаксис определения классов, рассматриваемый в главе 9) был расширен, чтобы сделать возможным сокращение, где ключевое слово `function` и двоеточие опускаются, давая в результате такой код:

```
let square = {
   area() { return this.side * this.side; },
   side: 10
};
square.area()    // => 100
```

Обе формы кода эквивалентны: обе добавляют в объектный литерал свойство по имени `area` и обе устанавливают значение этого свойства в указанную функцию. Сокращенный синтаксис делает яснее тот факт, что `area()` является методом, а не свойством с данными вроде `side`.

Когда вы записываете метод, используя сокращенный синтаксис, имя свойства может принимать любую форму, допустимую в объектном литерале: помимо обыкновенного идентификатора JavaScript наподобие `area` вы также можете применять строковые литералы и вычисляемые имена свойств, которые могут включать символьные имена свойств:

```
const METHOD_NAME = "m";
const symbol = Symbol();
let weirdMethods = {
    "method With Spaces"(x) { return x + 1; },
    [METHOD_NAME](x) { return x + 2; },
    [symbol](x) { return x + 3; }
};
weirdMethods["method With Spaces"](1)    // => 2
weirdMethods[METHOD_NAME](1)             // => 3
weirdMethods[symbol](1)                  // => 4
```

Использование значения Symbol в качестве имени метода не настолько странно, как кажется. Чтобы сделать объект итерируемым (что позволит его применять с циклом for/of), вы обязаны определить метод с символьным именем Symbol.iterator; примеры будут показаны в главе 12.

6.10.6. Методы получения и установки свойств

Все свойства объектов, которые обсуждались до сих пор, были *свойствами с данными*, имеющими имена и обыкновенные значения. В JavaScript также поддерживаются *свойства с методами доступа*, которые не имеют одиночного значения, но взамен располагают одним или двумя методами доступа: методом *получения* и/или методом *установки*.

Когда программа запрашивает значение свойства с методами доступа, интерпретатор JavaScript вызывает метод получения (без аргументов). Возвращаемое этим методом значение становится значением выражения доступа к свойству. Когда программа устанавливает значение свойства с методами доступа, интерпретатор JavaScript вызывает метод установки, передавая ему значение с правой стороны присваивания. В некотором смысле этот метод отвечает за "установку" значения свойства. Возвращаемое значение метода установки игнорируется.

Если свойство имеет метод получения и метод установки, то оно является свойством для чтения/записи. При наличии только метода получения свойство поддерживает только чтение. А если свойство располагает только методом установки, тогда оно считается свойством только для записи (в свойствах с данными подобное невозможно), и попытки его чтения всегда дают undefined.

Свойства с методами доступа можно определять посредством расширения синтаксиса объектных литералов (в отличие от рассмотренных здесь других расширений ES6 методы получения и установки появились в ES5):

```
let o = {
    // Обыкновенное свойство с данными
    dataProp: value,

    // Свойство с методами доступа определяется как пара функций
    get accessorProp() { return this.dataProp; },
    set accessorProp(value) { this.dataProp = value; }
};
```

Свойства с методами доступа определяются как один или два метода, имена которых совпадают с именем свойства. Они выглядят похожими на обычные методы, определенные с использованием сокращенной записи ES6, за исключением того, что определения методов получения и установки предваряются `get` или `set`. (При определении методов получения и установки в ES6 вы также можете применять вычисляемые имена свойств, для чего просто замените имя свойства после `get` или `set` выражением в квадратных скобках.)

Методы доступа, определенные ранее, лишь получали и устанавливали значение свойства с данными, поэтому не было особого смысла отдавать предпочтение свойству с методами доступа перед свойством с данными. Но в качестве более интересного примера рассмотрим следующий объект, который представляет двумерную декартову точку. Он имеет обыкновенные свойства с данными для представления координат *x* и *y* точки плюс свойства с методами доступа, которые дают эквивалентные полярные координаты точки:

```
let p = {
  //x и y - обыкновенные свойства с данными, допускающие чтение и запись
  x: 1.0,

  y: 1.0,
  // r - свойство с методами доступа, допускающее чтение и запись,
  // с методами получения и установки.
  // Не забывайте помещать запятую после методов доступа.
  get r() { return Math.hypot(this.x, this.y); },
  set r(newvalue) {
      let oldvalue = Math.hypot(this.x, this.y);
      let ratio = newvalue/oldvalue;
      this.x *= ratio;
      this.y *= ratio;
  },

  // theta - свойство с методами доступа, допускающее только чтение
  // и имеющее лишь метод получения.
  get theta() { return Math.atan2(this.y, this.x); }
};
p.r        // => Math.SQRT2
p.theta    // => Math.PI / 4
```

Обратите внимание в примере на использование ключевого слова `this` в методах получения и методе установки. Интерпретатор JavaScript вызывает эти функции как методы объекта, в котором они определены, т.е. внутри тела функции `this` ссылается на объект точки `p`. Таким образом, метод получения для свойства `r` может ссылаться на свойства `x` и `y` с помощью `this.x` и `this.y`. Методы и ключевое слово `this` будут детально раскрыты в подразделе 8.2.2.

Свойства с методами доступа наследуются в точности как свойства с данными, а потому вы можете применять определенный ранее объект `p` в качестве прототипа для других точек. Вы можете предоставить новым объектам собственные свойства `x` и `y`, а свойства `r` и `theta` они унаследуют:

```
let q = Object.create(p);// Новый объект, который наследует методы
                         // получения и установки
q.x = 3; q.y = 4;        // Создать в q собственные свойства с данными
q.r                      // => 5: унаследованные свойства
                         //       с методами доступа работают
q.theta                  // => Math.atan2(4, 3)
```

В показанном выше коде свойства с методами доступа используются для определения API-интерфейса, который обеспечивает два представления (декартовы и полярные координаты) одиночного набора данных. Другие причины применения свойств с методами доступа включают контроль корректности при записи свойств и возвращение отличающихся значений при чтении каждого свойства:

```
// Этот объект генерирует строго увеличивающиеся порядковые номера
const serialnum = {
    // Свойство с данными, которое хранит следующий порядковый номер.
    // Символ _ в имени свойства подсказывает, что оно предназначено
    // только для внутреннего использования.
    _n: 0,

    // Возвратить текущее значение и инкрементировать его
    get next() { return this._n++; },

    // Установить новое значение n, но только если оно больше текущего
    set next(n) {
        if (n > this._n) this._n = n;
        else throw new Error("порядковый номер можно устанавливать
только в большее значение");
    }
};
serialnum.next = 10;     // Установить начальный порядковый номер
serialnum.next           // => 10
serialnum.next           // => 11: при каждом обращении к next
                         //       мы получаем отличающееся значение
```

Наконец, ниже приведен еще один пример использования метода получения для реализации свойства с "магическим" поведением:

```
// Этот объект имеет свойства с методами доступа,
// которые возвращают случайные числа.
// Скажем, каждое вычисление выражения random.octet в результате
// дает случайное число между 0 и 255.
const random = {
    get octet() { return Math.floor(Math.random()*256); },
    get uint16() { return Math.floor(Math.random()*65536); },
    get int16() { return Math.floor(Math.random()*65536)-32768; }
};
```

6.11. Резюме

В главе были подробно описаны объекты JavaScript и раскрыты следующие основные темы.

- Базовая терминология, связанная с объектами, включая смысл терминов перечислимый и собственное свойство.
- Синтаксис объектных литералов, в том числе многие новые возможности, появившиеся в ES6 и последующих версиях.
- Способы чтения, записи, удаления, перечисления и проверки на предмет существования свойств объекта.
- Особенности работы наследования, основанного на прототипах, в JavaScript и создание объекта, унаследованного от другого объекта, с помощью `Object.create()`.
- Способы копирования свойств из одного объекта в другой с применением `Object.assign()`.

Все значения JavaScript, не относящиеся к элементарным значениям, являются объектами. Это касается массивов и функций, которые будут рассматриваться в следующих двух главах.

ГЛАВА 7

Массивы

В этой главе обсуждаются массивы — фундаментальный тип данных в JavaScript и большинстве других языков программирования. *Массив* представляет собой упорядоченную коллекцию значений. Каждое значение называется *элементом*, и каждый элемент имеет числовую позицию в массиве, известную как *индекс*. Массивы JavaScript являются *нетипизированными*: элемент массива может относиться к любому типу, а разные элементы одного массива могут иметь отличающиеся типы. Элементы массива могут быть даже объектами или другими массивами, что позволяет создавать сложные структуры данных, такие как массивы объектов и массивы массивов. Индексы в массивах JavaScript *начинаются с нуля* и представляют собой 32-битные числа: индекс первого элемента равен 0, а наибольший возможный индекс составляет 4294967294 ($2^{32}-2$) для максимального размера массива в 4 294 967 295 элементов. Массивы JavaScript являются *динамическими*: по мере надобности они увеличиваются или уменьшаются, а при создании массива нет необходимости объявлять для него фиксированный размер или повторно размещать в памяти в случае изменения его размера. Массивы JavaScript могут быть *разреженными*: элементы не обязаны иметь смежные индексы, поэтому возможно наличие брешей. Каждый массив JavaScript имеет свойство `length`. Для неразреженных массивов свойство `length` указывает количество элементов в массиве. Для разреженных массивов значение `length` больше самого высокого индекса любого элемента.

Массивы JavaScript являются специализированной формой объекта JavaScript, и в действительности индексы массивов — не многим более чем имена свойств, которые просто оказались целыми числами. О специализациях массивов речь пойдет в другом месте главы. Реализации обычно оптимизируют массивы, так что доступ к элементам массива с числовой индексацией, как правило, выполняется гораздо быстрее, нежели доступ к обыкновенным свойствам объектов.

Массивы наследуют свойства от `Array.prototype`, который определяет богатый набор методов манипулирования массивами, раскрываемый в разделе 7.8. Большинство этих методов являются *обобщенными*, а значит, они корректно работают не только с подлинными массивами, но также с любым "объектом, похожим на массив". Мы обсудим объекты, похожие на массивы, в разделе 7.9. Наконец, строки JavaScript ведут себя подобно массивам символов, и мы посмотрим на такое поведение в разделе 7.10.

В версии ES6 был введен набор новых классов типа массивов, коллективно известных как "типизированные массивы". В отличие от нормальных массивов JavaScript типизированные массивы имеют фиксированную длину и фиксированный числовой тип элементов. Они обеспечивают высокую производительность и доступ на уровне байтов к двоичным данным, как будет показано в разделе 11.2.

7.1. Создание массивов

Создавать массивы можно несколькими способами. В последующих подразделах приводятся объяснения, как создавать массивы с помощью:

- литералов типа массивов;
- операции распространения `...` на итерируемом объекте;
- конструктора `Array()`;
- фабричных методов `Array.of()` и `Array.from()`.

7.1.1. Литералы типа массивов

Пожалуй, самый простой способ создания массива предусматривает использование литерала типа массива, который представляет собой список элементов массива, разделенных запятыми, внутри квадратных скобок. Вот пример:

```
let empty = [];                 // Массив без элементов
let primes = [2, 3, 5, 7, 11];  // Массив с пятью числовыми элементами
let misc = [ 1.1, true, "a", ]; // Три элемента различных типов
                                // плюс хвостовая запятая
```

Значения литерала типа массива не обязательно должны быть константами; они могут быть произвольными выражениями:

```
let base = 1024;
let table = [base, base+1, base+2, base+3];
```

Литералы типа массивов могут содержать объектные литералы или другие литералы типа массивов:

```
let b = [[1, {x: 1, y: 2}], [2, {x: 3, y: 4}]];
```

Если литерал типа массива в какой-то строке содержит множество запятых без значений между ними, тогда массив будет разреженным (см. раздел 7.3). Элементы массива, для которых значения опущены, не существуют, но в случае их запрашивания представляются как undefined:

```
let count = [1,,3];         // Элементы находятся по индексам 0 и 2.
                            // По индексу 1 элемента нет
let undefs = [,,];          // Массив, не содержащий элементов,
                            // но имеющий длину 2
```

Синтаксис литералов типа массивов допускает наличие дополнительной хвостовой запятой, так что `[,,]` имеет длину 2, а не 3.

7.1.2. Операция распространения

В ES6 и более новых версиях можно применять "операцию распространения" `...`, чтобы помещать элементы одного массива внутрь литерала типа массива:

```
let a = [1, 2, 3];
let b = [0, ...a, 4];   // b == [0, 1, 2, 3, 4]
```

Многоточие "распространяет" массив `a`, так что его элементы становятся элементами создаваемого литерала типа массива. Результат получается такой же, как если бы конструкция `...a` была заменена элементами массива `a`, перечисленными буквально внутри включающего литерала типа массива. (Обратите внимание, что хотя мы называем многоточие операцией распространения, это не подлинная операция, поскольку ее можно использовать только в литералах типа массивов и, как будет показано позже в книге, в вызовах функций.)

Операция распространения обеспечивает удобный способ создания (неглубокой) копии массива:

```
let original = [1,2,3];
let copy = [...original];
copy[0] = 0;   // Модификация копии не приводит к изменению оригинала
original[0]    // => 1
```

Операция распространения работает с любым итерируемым объектом. (*Итерируемые* объекты — такие объекты, по которым выполняет проход цикл `for/of`; впервые они встречались в подразделе 5.4.4, а в главе 12 они будут рассматриваться более подробно.) Строки итерируемы, поэтому вы можете применять операцию распространения для превращения любой строки в массив односимвольных строк:

```
let digits = [..."0123456789ABCDEF"];
digits        // => ["0","1","2","3","4","5","6","7","8","9",
              //     "A","B","C","D","E","F"]
```

Объекты множеств `Set` (см. подраздел 11.1.1) итерируемы, так что легкий способ удаления дублированных элементов из массива предусматривает его преобразование в объект множества и затем немедленное преобразование множества в массив с использованием операции распространения:

```
let letters = [..."hello world"];
[...new Set(letters)]  // => ["h","e","l","o"," ","w","r","d"]
```

7.1.3. Конструктор `Array()`

Еще один способ создания массива — применение конструктора `Array()`, который можно вызывать тремя путями.

- Вызов без аргументов:

  ```
  let a= new Array();
  ```

  Такая методика создает пустой массив без элементов и эквивалентен литералу типа массива `[]`.

- Вызов с одиночным числовым аргументом, который указывает длину:

  ```
  let a= new Array(10);
  ```

  Такая методика создает массив с указанной длиной. Эту форму конструктора `Array()` можно использовать для предварительного размещения массива, когда заранее известно, сколько элементов потребуется. Важно отметить, что в результирующем массиве нет никаких значений, а свойства индексов массива "0", "1" и т.д. даже не определены.

- Явное указание двух и более элементов массива или одиночного нечислового элемента:

  ```
  let a = new Array(5, 4, 3, 2, 1, "testing, testing");
  ```

  В такой форме аргументы конструктора становятся элементами нового массива. Применять литерал типа массива почти всегда проще, чем конструктор `Array()` в подобной манере.

7.1.4. `Array.of()`

Когда функция конструктора `Array()` вызывается с одним числовым аргументом, она использует его как длину массива. Но при вызове с большим количеством числовых аргументов функция конструктора `Array()` трактует их как элементы для создаваемого массива. Это означает, что конструктор `Array()` не может применяться для создания массива с единственным числовым элементом.

В версии ES6 функция `Array.of()` решает описанную проблему: она представляет собой фабричный метод, который создает и возвращает новый массив, используя значения своих аргументов (независимо от их количества) в качестве элементов массива:

```
Array.of() //=>[]; при вызове без аргументов возвращает пустой массив
Array.of(10)       // => [10]; при вызове с единственным числовым
                   //          аргументом способна создавать массивы
Array.of(1,2,3)    // => [1, 2, 3]
```

7.1.5. `Array.from()`

`Array.from` — еще один фабричный метод, введенный в ES6. Он ожидает в первом аргументе итерируемый или похожий на массив объект и возвращает новый массив, который содержит элементы переданного объекта. С итерируемым аргументом `Array.from(iterable)` работает подобно операции распространения `[...iterable]`. Кроме того, она предлагает простой способ для получения копии массива:

```
let copy = Array.from(original);
```

Метод `Array.from()` также важен оттого, что он определяет способ создания копии в виде подлинного массива из объекта, похожего на массив. Объекты, похожие на массивы, не являются объектами массивов, которые имеют числовое свойство длины и значения, сохраненные с помощью свойств с именами в форме целых чисел. При написании кода JavaScript на стороне клиента возвращаемые значения некоторых методов веб-браузеров похожи на массивы, и

обрабатывать такие значения может быть легче, если сначала преобразовать их в подлинные массивы:

```
let truearray = Array.from(arraylike);
```

Метод `Array.from()` принимает необязательный второй аргумент. Если вы передадите во втором аргументе функцию, тогда в ходе построения нового массива каждый элемент из исходного объекта будет передаваться указанной вами функции, а возвращаемое ею значение будет сохраняться в массиве вместо первоначального значения. (Метод `Array.from()` во многом похож на метод `map()` массива, который будет представлен позже в главе, но выполнять сопоставление эффективнее во время построения массива, чем создать массив и затем отобразить его на другой новый массив.)

7.2. Чтение и запись элементов массива

Доступ к элементу массива осуществляется с применением операции `[]`. Ссылка на массив должна находиться слева от квадратных скобок. Внутрь квадратных скобок должно быть помещено произвольное выражение, имеющее неотрицательное целочисленное значение. Указанный синтаксис можно использовать как для чтения, так и для записи значения элемента массива. Таким образом, все нижеследующие операторы будут допустимыми в JavaScript:

```
let a = ["world"];      // Начать с одноэлементного массива
let value = a[0];       // Прочитать элемент 0
a[1] = 3.14;            // Записать элемент 1
let i = 2;
a[i] = 3;               // Записать элемент 2
a[i + 1] = "hello";     // Записать элемент 3
a[a[i]] = a[0];         // Прочитать элементы 0 и 2, записать элемент 3
```

Особенность массивов в том, что в случае применения имен свойств, которые являются неотрицательными целыми числами меньше $2^{32}-1$, массив автоматически поддерживает значение свойства `length`. Например, в предыдущем коде мы создали массив a с единственным элементом. Затем мы присвоили значения по индексам 1, 2 и 3. В итоге свойство `length` изменилось:

```
a.length                // => 4
```

Вспомните, что массивы — специализированный вид объекта. Квадратные скобки, используемые для доступа к элементам массива, работают в точности как квадратные скобки, применяемые для доступа к свойствам массива. Интерпретатор JavaScript преобразует указанный вами числовой индекс массива в строку — индекс 1 становится строкой `"1"`, — после чего использует результирующую строку в качестве имени свойства. В преобразовании индекса из числа в строку нет ничего особенного: вы можете поступать так и с обычными объектами:

```
let o = {};             // Создать простой объект
o[1] = "one";           // Индексировать его с помощью целого числа
o["1"]                  // => "one"; числовые и строковые имена свойств
                        //          считаются одинаковыми
```

Полезно четко отличать *индекс массива* от *имени свойства объекта*. Все индексы являются именами свойств, то только имена свойств, которые представляют собой целые числа между 0 и $2^{32}-2$, будут индексами. Все массивы являются объектами, и вы можете создавать в них свойства с любыми именами. Однако если вы применяете свойства, которые представляют собой индексы массива, то массивы поддерживают особое поведение по обновлению их свойства `length` по мере необходимости.

Обратите внимание, что индексировать массив можно с использованием отрицательных или нецелых чисел. Такое число преобразуется в строку, которая применяется в качестве имени свойства. Поскольку имя свойства не является неотрицательным целым числом, оно трактуется как обыкновенное свойство объекта, а не индекс массива. Кроме того, при индексировании массива посредством строки, которая в результате преобразования дает неотрицательное целое число, она ведет себя как индекс массива, но не как свойство объекта. То же самое справедливо в случае использования числа с плавающей точкой, совпадающего по величине с целым числом:

```
a[-1.23] = true;    // Это создает свойство по имени "-1.23"
a["1000"] = 0;      // Это 1001-й элемент массива
a[1.000] = 1;       // Индекс массива 1. То же самое, что и a[1] = 1;
```

Тот факт, что индексы представляют собой просто особый тип имени свойства объекта, означает отсутствие в массивах JavaScript понятия ошибки "выхода за границы". При попытке запроса несуществующего свойства любого объекта вы получите не ошибку, а всего лишь `undefined`. Так происходит и с массивами, и с объектами:

```
let a = [true, false]; //Этот массив содержит элементы по индексам 0 и 1
a[2]      // => undefined; элемент по этому индексу отсутствует
a[-1]     // => undefined; свойство с таким именем не существует
```

7.3. Разреженные массивы

Разреженный массив — это массив, элементы которого не имеют непрерывных индексов, начинающихся с 0. Обычно свойство `length` массива указывает количество элементов в массиве. Когда массив разреженный, значение свойства `length` будет больше количества элементов. Разреженные массивы можно создавать с помощью конструктора `Array()` и просто за счет присваивания по индексу, превышающему текущее значение свойства `length` массива.

```
let a = new Array(5);   // Элементы отсутствуют, но a.length равно 5
a = [];                 // Создает массив без элементов и length = 0
a[1000] = 0;            // Присваивание добавляет один элемент,
                        // но устанавливает length в 1001
```

Позже вы увидите, что делать массив разреженным можно также посредством операции `delete`.

В достаточной степени разреженные массивы обычно реализуются медленным, более эффективным в плане памяти способом в сравнении с плотными

массивами и поиск элементов в таких массивах занимает почти столько же времени, сколько поиск обыкновенных свойств объектов.

Обратите внимание, что когда вы опускаете значение в литерале типа массива (применяя повторяющиеся запятые, как в `[1,,3]`), результирующий массив будет разреженным, а опущенные элементы просто не существуют:

```
let a1 = [,];           // Этот массив не содержит элементов
                        // и значение length равно 1
let a2 = [undefined];   // Этот массив имеет один элемент undefined
0 in a1             // => false: a1 не содержит элемента по индексу 0
0 in a2             // => true: a2 имеет значение undefined по индексу 0
```

Понимание разреженных массивов — важная часть понимания истинной природы массивов JavaScript. Тем не менее, на практике большинство массивов JavaScript, с которыми вам придется иметь дело, не будут разреженными. И если вам доведется работать с разреженным массивом, то вполне вероятно, что в коде вы будете обращаться с ним как с неразреженным массивом, содержащим элементы `undefined`.

7.4. Длина массива

Каждый массив имеет свойство `length`, и именно оно делает массивы отличающимися от обыкновенных объектов JavaScript. Для плотных (т.е. неразреженных) массивов свойство `length` указывает количество элементов. Его значение на единицу больше самого высокого индекса в массиве:

```
[].length               // => 0: массив не содержит элементов
["a","b","c"].length    // => 3: самый высокий индекс равен 2,
                        //       значение length равно 3
```

Когда массив разреженный, значение свойства `length` больше количества элементов, и мы можем сказать лишь то, что значение `length` гарантированно превышает индекс любого элемента в массиве. Либо, говоря по-другому, массив (разреженный или нет) никогда не будет иметь элемент, индекс которого больше или равен значению его свойства `length`. Для поддержания такого постоянства длины массивы обладают двумя специальными линиями поведения. Первая была описана выше: если вы присвоите значение элементу массива с индексом `i`, который больше или равен текущему значению свойства `length` массива, то свойство `length` установится в `i+1`.

Вторая специальная линия поведения, реализуемая массивами для поддержания постоянства длины, заключается в том, что если вы установите свойство `length` в неотрицательное целое число `n`, которое меньше текущего значения `length`, тогда любые элементы, чьи индексы больше или равны `n`, удалятся из массива:

```
a = [1,2,3,4,5];    // Начать с пятиэлементного массива
a.length = 3;       // a теперь [1,2,3]
a.length = 0;       // Удалить все элементы. a становится []
a.length = 5;       // Длина равна 5, но элементы отсутствуют,
                    // подобно new Array(5)
```

Вы также можете установить свойство length массива в значение, превышающее его текущее значение. Фактически это не приведет к добавлению каких-то новых элементов в массив, а просто создаст разреженную область в конце массива.

7.5. Добавление и удаление элементов массива

Ранее уже был показан простейший способ добавления элементов в массив — присваивание значений по новым индексам:

```
let a = [];              // Начать с пустого массива
a[0] = "zero";           // И добавить в него элементы
a[1] = "one";
```

Вы можете использовать метод push() для добавления одного и более значений в конец массива:

```
let a = [];              // Начать с пустого массива
a.push("zero");          // Добавить значение в конец. a = ["zero"]
a.push("one", "two");    // Добавить еще два значения.
                         //   a = ["zero", "one", "two"]
```

Заталкивание значения в массив a представляет собой то же самое, что и присваивание значения элементу a[a.length]. Вы можете применять метод unshift(), описанный в разделе 7.8, для вставки значения в начало массива со сдвигом существующих элементов, находящихся по более высоким индексам. Метод pop() — противоположность push(): он удаляет последний элемент массива, уменьшая его длину на 1. Аналогично метод shift() удаляет и возвращает первый элемент массива, уменьшая его длину на 1 и сдвигая все элементы по индексам, которые на единицу меньше их текущих индексов. Дополнительные сведения об указанных методах приведены в разделе 7.8.

Вы можете удалять элементы массива с помощью операции delete, в точности как удаляли свойства объектов:

```
let a = [1,2,3];
delete a[2];             // Теперь a не имеет элемента по индексу 2
2 in a                   // => false: индекс 2 в массиве не определен
a.length                 // => 3: delete не влияет на длину массива
```

Удаление элемента массива похоже на присваивание ему undefined (но имеет тонкое отличие). Обратите внимание, что использование операции delete с элементом массива не изменяет значение свойства length и не сдвигает элементы с более высокими индексами, чтобы заполнить брешь, оставленную удалением. Если вы удаляете элемент из массива, то он становится разреженным.

Как упоминалось ранее, удалять элементы из конца массива также можно, просто устанавливая свойство length в желаемую новую длину.

Наконец, splice() является универсальным методом для вставки, удаления или замены элементов массива. Он изменяет свойство length и необходимым образом сдвигает элементы массива в сторону более высоких или низких индексов. Детали ищите в разделе 7.8.

7.6. Итерация по массивам

Начиная с версии ES6, самый легкий способ циклического прохода по массиву (или любому итерируемому объекту) связан с применением цикла `for/of`, который подробно рассматривался в подразделе 5.4.4:

```
let letters = [..."Hello world"];  // Массив букв
let string = "";
for(let letter of letters) {
   string += letter;
}
string  // => "Hello world"; мы повторно собрали первоначальный текст
```

Встроенный итератор по массиву, который использует цикл `for/of`, возвращает элементы массива в возрастающем порядке. Никакого специального поведения в отношении разреженных массивов в нем не предусмотрено, и он просто возвращает `undefined` для всех несуществующих элементов массива.

Если вы хотите применять с массивом цикл `for/of` и знать индекс каждого элемента массива, тогда используйте метод `entries()` массива вместе с деструктурирующим присваиванием, например:

```
let everyother = "";
for(let [index, letter] of letters.entries()) {
   if (index % 2 === 0) everyother += letter; // Буквы по четным индексам
}
everyother // => "Hlowrd"
```

Еще один хороший способ итерации по массивам предполагает применение `forEach()`. Это не новая форма цикла `for`, а метод массива, который предлагает функциональный подход к итерации. Вы передаете методу `forEach()` массива функцию и `forEach()` будет вызывать указанную функцию по одному разу для каждого элемента в массиве:

```
let uppercase = "";
letters.forEach(letter => {// Обратите внимание на синтаксис
                           // стрелочной функции
   uppercase += letter.toUpperCase();
});
uppercase  // => "HELLO WORLD"
```

Как и можно было ожидать, `forEach()` организует итерацию по массиву в соответствующем порядке и на самом деле передает вашей функции во втором аргументе индекс массива, что иногда оказывается полезно. В отличие от цикла `for/of` метод `forEach()` осведомлен о разреженных массивах и не вызывает вашу функцию для несуществующих элементов.

Метод `forEach()` более подробно обсуждается в подразделе 7.8.1, где также раскрываются связанные методы, такие как `map()` и `filter()`, которые выполняют более специализированные виды итерации по массивам.

Вдобавок вы можете проходить по элементам массива с помощью старого доброго цикла `for` (см. подраздел 5.4.3):

```
let vowels = "";
for(let i = 0; i < letters.length; i++) {//Для каждого индекса в массиве
    let letter = letters[i];        // Получить элемент по этому индексу
    if (/[aeiou]/.test(letter)) { // Использовать проверку
                                  // с регулярным выражением
        vowels += letter;         // Если гласная буква, то запомнить ее
    }
}
vowels                            // => "eoo"
```

Во вложенных циклах или в других контекстах, где критически важна производительность, временами вы можете встречать приведенный базовый цикл итерации по массиву, записанный так, что длина массива ищется только раз, а не на каждой итерации. Обе показанные ниже формы цикла `for` являются идиоматичными, хотя не особо распространенными, и с современными интерпретаторами JavaScript не вполне ясно, оказывают ли они влияние на производительность:

```
// Сохранить длину массива в локальной переменной
for(let i = 0, len = letters.length; i < len; i++) {
    // тело цикла остается прежним
}

// Итерация в обратном направлении с конца до начала массива
for(let i = letters.length-1; i >= 0; i--) {
    // тело цикла остается прежним
}
```

В примерах предполагается, что массив плотный и все элементы содержат допустимые данные. Если это не так, тогда вы должны проверять элементы массива, прежде чем использовать их. Если нужно пропускать неопределенные и несуществующие элементы, то вы можете написать код следующего вида:

```
for(let i = 0; i < a.length; i++) {
    if (a[i] === undefined) continue;     // Пропускать неопределенные
                                          // и несуществующие элементы
    // тело цикла
}
```

7.7. Многомерные массивы

В JavaScript не поддерживаются подлинные многомерные массивы, но вы можете приблизиться к ним через массивы массивов. Для доступа к значению в массиве массивов просто дважды применяйте операцию `[]`. Пусть, например, переменная `matrix` представляет собой массив массивов чисел. Каждый элемент в `matrix[x]` является массивом чисел. Чтобы получить доступ к отдельному числу внутри этого массива, вы должны записать `matrix[x][y]`. Вот конкретный пример, в котором двумерный массив используется как таблица умножения:

```
// Создать многомерный массив
let table = new Array(10);          // 10 строк таблицы
for(let i = 0; i < table.length; i++) {
   table[i] = new Array(10);        // Каждая строка имеет 10 столбцов
}
// Инициализировать массив
for(let row = 0; row < table.length; row++) {
   for(let col = 0; col < table[row].length; col++) {
      table[row][col] = row*col;
   }
}
// Использовать многомерный массив для вычисления 5*7
table[5][7]  // => 35
```

7.8. Методы массивов

В предшествующих разделах внимание было сосредоточено на базовом синтаксисе JavaScript для работы с массивами. Однако в целом самые мощные методы определены в классе `Array`. Они будут описаны в последующих разделах. Имейте в виду, что некоторые из них модифицируют массив, на котором вызываются, тогда как другие оставляют массив неизмененным. Несколько методов возвращают массив: иногда это новый массив и оригинал не изменяется. В других случаях метод будет модифицировать массив на месте и возвращать ссылку на модифицированный массив.

В каждом из следующих далее подразделов раскрывается группа связанных методов массивов:

- методы итераторов проходят по элементам массива, обычно вызывая на каждом элементе функцию, которую вы указываете;
- методы стеков и очередей добавляют и удаляют элементы в начале и конце массива;
- методы подмассивов извлекают, удаляют, вставляют, заполняют и копируют непрерывные области более крупного массива;
- методы поиска и сортировки ищут элементы внутри массива и сортируют элементы в массиве.

В последующих подразделах также рассматриваются статические методы класса `Array` и несколько смешанных методов, предназначенных для объединения массивов и преобразования массивов в строки.

7.8.1. Методы итераторов для массивов

Описанные в настоящем разделе методы проходят по массивам, передавая элементы массива по очереди предоставленной вами функции, и предлагают удобные способы для итерации, отображения, проверки и сокращения массивов.

Тем не менее, прежде чем детально объяснять методы, полезно сделать о них некоторые обобщения. Все эти методы принимают функцию в своем первом ар-

гументе и вызывают ее по одному разу для каждого элемента (или ряда элементов) массива. Если массив разреженный, тогда передаваемая вами функция не вызывается для несуществующих элементов. В большинстве случаев функция, которую вы предоставляете, вызывается с тремя аргументами: значение элемента массива, индекс элемента массива и сам массив. Зачастую вас интересует только первое из трех значений аргументов, а потому второе и третье значения вы можете проигнорировать.

Большинство методов итераторов, документированных в последующих подразделах, принимают необязательный второй аргумент. Если он указан, то функция вызывается так, как будто является методом второго аргумента. То есть передаваемый вами второй аргумент становится значением ключевого слова `this` внутри функции, которую вы передали в качестве первого аргумента. Возвращаемое значение передаваемой функции обычно важно, но разные методы обрабатывают его по-разному. Ни один из описываемых здесь методов не модифицирует массив, на котором вызывается (хотя передаваемая вами функция, конечно же, может модифицировать массив).

Каждая из этих функций вызывается с функцией в своем первом аргументе, и такую функцию очень часто определяют внутристрочным образом как часть выражения вызова метода, а не применяют существующую функцию, определенную где-то в другом месте. Как будет показано позже в примерах, с такими методами довольно хорошо работает синтаксис стрелочных функций (см. подраздел 8.1.3).

forEach()

Метод `forEach()` проходит по массиву, вызывая для каждого элемента указанную вами функцию. Как уже было описано, вы передаете методу `forEach()` функцию в первом аргументе. Затем `forEach()` вызывает вашу функцию с тремя аргументами: значение элемента массива, индекс элемента массива и сам массив. Если вас заботит только значение элемента массива, тогда вы можете написать функцию с единственным параметром — добавочные аргументы будут проигнорированы:

```
let data = [1,2,3,4,5], sum = 0;
// Вычислить сумму элементов массива
data.forEach(value => { sum += value; });          // sum == 15

// А теперь инкрементировать каждый элемент массива
data.forEach(function(v, i, a) { a[i] = v + 1; });//data == [2,3,4,5,6]
```

Обратите внимание, что `forEach()` не предлагает какого-либо способа закончить итерацию до того, как все элементы будут переданы в функцию. То есть никакого эквивалента оператора `break`, который вы можете использовать с обыкновенным циклом `for`, не предусмотрено.

map()

Метод `map()` передает каждый элемент массива, на котором он вызван, указанной вами функции и возвращает массив со значениями, возвращенными вашей функцией. Например:

```
let a = [1, 2, 3];
a.map(x => x*x)          // => [1, 4, 9]: функция принимает на входе x
                         //                и возвращает x*x
```

Функция, передаваемая вами методу `map()`, вызывается таким же способом, как и функция, передаваемая `forEach()`. Однако для метода `map()` функция, которую вы передаете, должна возвращать значение. Следует отметить, что метод `map()` возвращает новый массив: он не модифицирует массив, на котором вызывается. Если исходный массив разреженный, тогда ваша функция не будет вызываться на недостающих элементах, но возвращаемый массив окажется разреженным в таком же стиле, как исходный: с той же самой длиной и теми же недостающими элементами.

`filter()`

Метод `filter()` возвращает массив, содержащий подмножество элементов массива, на котором он был вызван. Передаваемая вами функция обязана быть предикатом: функцией, которая возвращает `true` или `false`. Предикат вызывается в точности как для `forEach()` и `map()`. Если возвращаемое значение равно `true` или преобразуется в `true`, тогда элемент, переданный предикату, входит в подмножество и добавляется в массив, который станет возвращаемым значением. Вот примеры:

```
let a = [5, 4, 3, 2, 1];
a.filter(x => x < 3)          // => [2, 1]; значения меньше 3
a.filter((x,i) => i%2 === 0)  // => [5, 3, 1]; все остальные значения
```

Обратите внимание, что метод `filter()` пропускает недостающие элементы и возвращаемый им массив всегда плотный. Чтобы избавиться от брешей в разреженном массиве, вы можете поступить так:

```
let dense = sparse.filter(() => true);
```

А для устранения брешей и удаления элементов `undefined` и `null` вы можете применить `filter()` следующим образом:

```
a = a.filter(x => x !== undefined && x !== null);
```

`find()` и `findIndex()`

Методы `find()` и `findIndex()` похожи на `filter()` в том, что они проходят по массиву в поиске элементов, для которых ваша функция предиката возвращает истинное значение. Тем не менее, в отличие от `filter()` эти два метода останавливают итерацию, как только предикат впервые находит элемент. Когда такое происходит, `find()` возвращает совпадающий элемент, а `findIndex()` — индекс совпадающего элемента. Если совпадающий элемент не обнаружен, тогда `find()` возвращает `undefined`, а `findIndex()` — значение `-1`:

```
let a = [1,2,3,4,5];
a.findIndex(x => x === 3)    // => 2; значение 3 находится по индексу 2
a.findIndex(x => x < 0)      // => -1; отрицательных чисел в массиве нет
a.find(x => x % 5 === 0)     // => 5: число, кратное 5
a.find(x => x % 7 === 0)     // => undefined: числа, кратные 7,
                             //                в массиве отсутствуют
```

every() и some()

Методы every() и some() являются предикатами массивов: они применяют указанную вами функцию предиката к элементам массива, после чего возвращают true или false.

Метод every() подобен математическому квантору "для всех" (∀): он возвращает true, если и только если ваша функция предиката возвращает true для всех элементов в массиве:

```
let a = [1,2,3,4,5];
a.every(x => x < 10)        // => true: все значения < 10
a.every(x => x % 2 === 0)   // => false: не все значение четные
```

Метод some() подобен математическому квантору "существует" (∃): он возвращает true, если в массиве существует, по крайней мере, один элемент, для которого предикат возвращает true, и false, если и только если предикат возвращает false для всех элементов массива:

```
let a = [1,2,3,4,5];
a.some(x => x%2===0)    // => true; а содержит несколько четных чисел
a.some(isNaN)           // => false; а не содержит "не числа"
```

Следует отметить, что методы every() и some() останавливают итерацию по элементам массива, как только им становится известно, какое значение возвращать. Метод some() возвращает true, когда ваш предикат первый раз возвращает true, и полностью проходит массив, только если предикат всегда возвращает false. Метод every() является противоположностью: он возвращает false, когда ваш предикат первый раз возвращает false, и проходит по всем элементам, только если предикат всегда возвращает true. Кроме того, обратите внимание, что по математическому соглашению при вызове на пустом массиве метод every() возвращает true, а some() возвращает false.

reduce() и reduceRight()

Методы reduce() и reduceRight() объединяют элементы массива, используя указанную вами функцию, для получения единственного значения. Такая операция распространена в функциональном программировании и встречается под названиями "внедрение" и "свертка". Ниже в примерах иллюстрируется ее работа:

```
let a = [1,2,3,4,5];
a.reduce((x,y) => x+y, 0)           // => 15; сумма значений
a.reduce((x,y) => x*y, 1)           // => 120; произведение значений
a.reduce((x,y) => (x > y) ? x : y)  // => 5; наибольшее из значений
```

Метод reduce() принимает два аргумента. Первый — это функция, которая выполняет операцию сокращения. Задача функции сокращения заключается в том, чтобы каким-то образом объединить или сократить два значения в единственное значение и возвратить такое сокращенное значение. В показанных здесь примерах функции объединяли два значения, складывая их, перемножая их и выбирая наибольшее из них. Во втором (необязательном) аргументе указывается начальное значение для передачи функции.

Функции, применяемые с reduce(), отличаются от тех, которые используются с forEach() и map(). Уже знакомые значение, индекс и массив передаются как второй, третий и четвертый аргументы. Первый аргумент представляет собой накопленный до сих пор результат сокращения. При первом вызове функции первый аргумент содержит начальное значение, которое вы передавали во втором аргументе методу reduce(). При последующих вызовах первый аргумент будет значением, возвращенным предыдущим вызовом функции. В первом примере функция сокращения сначала вызывается с аргументами 0 и 1. Она складывает их и возвращает 1. Затем она вызывается снова с аргументами 1 и 2 и возвращает 3. Далее функция вычисляет 3+3=6, потом 6+4=10 и в заключение 10+5=15. Финальное значение 15 становится возвращаемым значение метода reduce().

Вы могли также заметить, что третий вызов reduce() в примере имеет только один аргумент: начальное значение не указывалось. Когда метод reduce() вызывается без начального значения, он использует в качестве него первый элемент массива. Таким образом, первый вызов функции сокращения получит в своих первом и втором аргументах первый и второй элементы массива. В примерах с суммой и произведением значений мы тоже могли бы опустить аргумент начального значения.

Вызов метода reduce() на пустом массиве без аргумента начального значения приводит к генерации TypeError. Если вы вызовите его с только одним значением — либо массивом с одним элементом и без начального значения, либо с пустым массивом и начальным значением, — тогда он просто возвратит это одно значение, не вызывая функцию сокращения.

Метод reduceRight() работает в точности как reduce(), но обрабатывает массив от самого высокого индекса до самого низкого (справа налево), а не от самого низкого до самого высокого. Вы могли бы задействовать его, например, если операция сокращения имеет ассоциативность справа налево:

```
// Вычисляет 2^(3^4). Операция возведения в степень
// имеет ассоциативность справа налево
let a = [2, 3, 4];
a.reduceRight((acc,val)=>Math.pow(val,acc)) //=>2.4178516392292583e+24
```

Обратите внимание, что ни reduce(), ни reduceRight() не принимают необязательный аргумент, указывающий значение this, на котором должна вызываться функция сокращения. Его место занимает необязательный аргумент с начальным значением. Если вам необходимо, чтобы функция сокращения вызывалась как метод определенного объекта, тогда взгляните на метод Function.bind() (см. подраздел 8.7.5).

Ради простоты показанные до сих пор примеры были числовыми, но reduce() и reduceRight() предназначены не только для математических расчетов. В качестве функции сокращения может применяться любая функция, которая способна объединять два значения (такие как два объекта) в одно значение того же самого типа. С другой стороны, алгоритмы, выраженные с использованием сокращения массивов, быстро становятся сложными и трудными для понимания, поэтому вы можете обнаружить, что ваш код легче читать,

писать и осмысливать, если для обработки массивов в нем будут применяться обыкновенные конструкции циклов.

7.8.2. Выравнивание массивов с помощью `flat()` и `flatMap()`

В ES2019 метод `flat()` создает и возвращает новый массив, содержащий те же самые элементы, что и массив, на котором он вызывался, но любые элементы, являющиеся массивами, в результирующем массиве будут "выровненными". Вот пример:

```
[1, [2, 3]].flat()              // => [1, 2, 3]
[1, [2, [3]]].flat()            // => [1, 2, [3]]
```

В случае вызова без аргументов метод `flat()` выравнивает один уровень вложения. Элементы первоначального массива, которые сами представляют собой массивы, выравниваются, но элементы *этих* массивов не выравниваются. Если вы хотите выровнять больше уровней, тогда передайте методу `flat()` число:

```
let a = [1, [2, [3, [4]]]];
a.flat(1)                       // => [1, 2, [3, [4]]]
a.flat(2)                       // => [1, 2, 3, [4]]
a.flat(3)                       // => [1, 2, 3, 4]
a.flat(4)                       // => [1, 2, 3, 4]
```

Метод `flatMap()` работает точно как метод `map()` (см. раздел "`map()`" ранее в главе) за исключением того, что возвращаемый массив автоматически выравнивается, как если бы он передавался `flat()`. То есть вызов `a.flatMap(f)` — то же самое, что и `a.map(f).flat()`, но эффективнее:

```
let phrases = ["hello world", "the definitive guide"];
let words = phrases.flatMap(phrase => phrase.split(" "));
words  // => ["hello", "world", "the", "definitive", "guide"];
```

Вы можете считать `flatMap()` обобщением `map()`, которое позволяет отображать каждый элемент входного массива на любое количество элементов выходного массива. В частности, `flatMap()` дает возможность отобразить входные элементы на пустой массив, который ничего не выравнивает в выходном массиве:

```
// Отобразить неотрицательные числа на их квадратные корни
[-2, -1, 1, 2].flatMap(x => x < 0 ? [] : Math.sqrt(x)) // => [1, 2**0.5]
```

7.8.3. Присоединение массивов с помощью `concat()`

Метод `concat()` создает и возвращает новый массив, содержащий элементы исходного массива, за которыми следуют все аргументы, переданные `concat()`. Если любой из аргументов является массивом, то присоединяются его элементы, но не сам массив. Однако имейте в виду, что `concat()` не выравнивает рекурсивно массивы массивов. Метод `concat()` не модифицирует массив, на котором он вызывается:

```
let a = [1,2,3];
a.concat(4, 5)          // => [1,2,3,4,5]
a.concat([4,5],[6,7])   // => [1,2,3,4,5,6,7]; массивы выравниваются
a.concat(4, [5,[6,7]]) // =>[1,2,3,4,5,[6,7]];но не вложенные массивы
a                       // => [1,2,3]; исходный массив остается
                        //              немодифицированным
```

Обратите внимание, что `concat()` создает новую копию массива, на котором вызывается. Во многих случаях поступать так правильно, но операция оказывается затратной. Если вы обнаруживаете, что пишете код вроде `a = a.concat(x)`, то должны подумать о модификации своего массива на месте посредством `push()` или `splice()`, а не создавать новый массив.

7.8.4. Организация стеков и очередей с помощью `push()`, `pop()`, `shift()` и `unshift()`

Методы `push()` и `pop()` позволяют работать с массивами, как если бы они были стеками. Метод `push()` добавляет один или большее количество элементов в конец массива и возвращает новую длину массива. В отличие от `concat()` метод `push()` не выравнивает аргументы типа массивов. Метод `pop()` выполняет противоположное действие: он удаляет последний элемент массива, декрементирует длину массива и возвращает значение, которое было удалено. Обратите внимание, что оба метода модифицируют массив на месте. Сочетание `push()` и `pop()` дает возможность использовать массив JavaScript для реализации стека магазинного типа. Например:

```
let stack = [];          // stack == [] `
stack.push(1,2);         // stack == [1,2];
stack.pop();             // stack == [1]; возвращается 2
stack.push(3);           // stack == [1,3]
stack.pop();             // stack == [1]; возвращается 3
stack.push([4,5]);       // stack == [1,[4,5]]
stack.pop()              // stack == [1]; возвращается [4,5]
stack.pop();             // stack == []; возвращается 1
```

Метод `push()` не выравнивает передаваемый ему массив, но если вы хотите поместить все элементы одного массива в другой, то можете применить операцию распространения (см. подраздел 8.3.4), чтобы выровнять его неявно:

```
a.push(...values);
```

Методы `unshift()` и `shift()` ведут себя во многом похоже на `push()` и `pop()` за исключением того, что они вставляют и удаляют элементы с начала, а не с конца массива. Метод `unshift()` добавляет элемент или элементы в начало массива, сдвигая существующие элементы в направлении более высоких индексов с целью освобождения пространства, и возвращает новую длину массива. Метод `shift()` удаляет и возвращает первый элемент массива, сдвигая все последующие элементы на одну позицию в сторону уменьшения индексов, чтобы занять освободившееся место в начале массива. Вы могли бы использовать `unshift()` и `shift()` для реализации стека, но это было бы менее эффективно, чем применение `push()` и `pop()`, поскольку каждый раз, когда в начале

массива добавляется или удаляется элемент, существующие элементы нужно сдвигать в одном или другом направлении. Но взамен вы можете реализовать структуру данных типа очереди за счет использования `push()` для добавления элементов в конец массива и `shift()` для их удаления с начала массива:

```
let q = [];             // q == []
q.push(1,2);            // q == [1,2]
q.shift();              // q == [2]; возвращается 1
q.push(3)               // q == [2, 3]
q.shift()               // q == [3]; возвращается 2
q.shift()               // q == []; возвращается 3
```

Существует одна особенность метода `unshift()`, о которой стоит упомянуть, т.к. она может вызвать у вас удивление. В случае передачи `unshift()` множества аргументов они вставляются все сразу, а это значит, что они окажутся в массиве не в том порядке, в каком они были бы при вставке их по одному за раз:

```
let a = [];             // a == []
a.unshift(1)            // a == [1]
a.unshift(2)            // a == [2, 1]
a = [];                 // a == []
a.unshift(1,2)          // a == [1, 2]
```

7.8.5. Работа с подмассивами с помощью `slice()`, `splice()`, `fill()` и `copyWithin()`

В классе `Array` определено несколько методов, которые работают на последовательных областях, или подмассивах либо "срезах" массива. В следующих далее подразделах будут описаны методы для извлечения, замены, заполнения и копирования срезов.

`slice()`

Метод `slice()` возвращает *срез*, или подмассив, заданного массива. В двух его аргументах указываются начало и конец среза, подлежащего возвращению. Возвращенный массив содержит элемент, указанный первым аргументом, и все последующие элементы вплоть до элемента, указанного вторым аргументом (не включая его). Если указан только один аргумент, тогда возвращенный массив содержит все элементы от начальной позиции до конца массива. Если любой из двух аргументов отрицательный, то он задает элемент относительно длины массива. Скажем, аргумент –1 указывает на последний элемент в массиве, а аргумент –2 — на элемент перед последним элементом. Обратите внимание, что метод `slice()` не модифицирует массив, на котором вызывается. Вот несколько примеров:

```
let a = [1,2,3,4,5];
a.slice(0,3);           // Возвращается [1,2,3]
a.slice(3);             // Возвращается [4,5]
a.slice(1,-1);          // Возвращается [2,3,4]
a.slice(-3,-2);         // Возвращается [3]
```

`splice()`

`splice()` — это универсальный метод для вставки или удаления элементов из массива. В отличие от `slice()` и `concat()` метод `splice()` модифицирует

массив, на котором вызывается. Важно отметить, что splice() и slice() имеют очень похожие имена, но реализуют существенно отличающиеся действия.

Метод splice() способен удалять элементы из массива, вставлять новые элементы в массив или выполнять оба действия одновременно. Индексы элементов массива, находящихся после точки вставки или удаления, должным образом увеличиваются или уменьшаются, так что они остаются смежными с остатком массива. Первый аргумент splice() задает позицию в массиве, где начинается вставка и/или удаление. Второй аргумент указывает количество элементов, которые требуется удалить из (отщепить от) массива. (Обратите внимание, что это еще одно отличие между методами splice() и slice(). Во втором аргументе slice() передается конечная позиция, а во втором аргументе splice() — длина.) Если второй аргумент опущен, тогда удаляются все элементы массива от начального элемента и до конца массива. Метод splice() возвращает массив с удаленными элементами или пустой массив, если элементы не были удалены. Например:

```
let a = [1,2,3,4,5,6,7,8];
a.splice(4)             // => [5,6,7,8]; a теперь [1,2,3,4]
a.splice(1,2)           // => [2,3]; a теперь [1,4]
a.splice(1,1)           // => [4]; a теперь [1]
```

Первые два аргумента splice() указывают, какие элементы массива подлежат удалению. За этими аргументами может следовать любое количество дополнительных аргументов, задающих элементы, которые должны быть вставлены в массив, начиная с позиции, указанной в первом аргументе. Вот пример:

```
let a = [1,2,3,4,5];
a.splice(2,0,"a","b")  // => []; a теперь [1,2,"a","b",3,4,5]
a.splice(2,2,[1,2],3)  // => ["a","b"]; a теперь [1,2,[1,2],3,3,4,5]
```

Следует отметить, что в отличие от concat() метод splice() вставляет сами массивы, а не их элементы.

fill()

Метод fill() устанавливает элементы массива или среза массива в указанное значение. Он видоизменяет массив, на котором вызывается, и также возвращает модифицированный массив:

```
let a = new Array(5);   // Начать с массива без элементов длиной 5
a.fill(0)               // => [0,0,0,0,0]; заполнить массив нулями
a.fill(9, 1)            // => [0,9,9,9,9]; заполнить значениями 9,
                        //                 начиная с индекса 1
a.fill(8, 2, -1)        // => [0,9,8,8,9]; заполнить значениями 8
                        //                 по индексам 2, 3
```

Первый аргумент fill() представляет собой значение, в которое будут установлены элементы массива. В необязательном втором аргументе указывается начальный индекс. Если второй аргумент опущен, тогда заполнение начинается с индекса 0. В необязательном третьем аргументе задается конечный индекс — элементы массива будут заполняться вплоть до этого индекса, не включая его. Если третий аргумент опущен, то массив заполняется от начального индекса до конца. Как и в slice(), вы можете указывать индексы относительно конца массива, передавая отрицательные числа.

copyWithin()

Метод copyWithin() копирует срез массива в новую позицию внутри массива. Он модифицирует массив на месте и возвращает модифицированный массив, но не изменяет длину массива. В первом аргументе задается целевой индекс, по которому будет копироваться первый элемент. Во втором аргументе указывается индекс первого копируемого элемента. Если второй аргумент опущен, тогда применяется 0. В третьем аргументе задается конец среза элементов, подлежащих копированию. Если третий аргумент опущен, то используется длина массива. Копироваться будут элементы от начального индекса и вплоть до конечного индекса, не включая его. Как и в slice(), вы можете указывать индексы относительно конца массива, передавая отрицательные числа:

```
let a = [1,2,3,4,5];
a.copyWithin(1)    // => [1,1,2,3,4]: копировать элементы массива
                   //                 в позиции, начиная с первого
a.copyWithin(2, 3, 5)  // => [1,1,3,4,4]: копировать последние
                       //                 2 элемента по индексу 2
a.copyWithin(0, -2)    // => [4,4,3,4,4]: отрицательные смещения
                       //                 тоже работают
```

Метод copyWithin() задумывался как высокопроизводительный метод, который особенно полезен при работе с типизированными массивами (см. раздел 11.2). Он моделирует функцию memmove() из стандартной библиотеки C. Обратите внимание, что копирование будет корректно работать, даже если исходная и целевая области перекрываются.

7.8.6. Методы поиска и сортировки массивов

В массивах реализованы методы indexOf(), lastIndexOf() и includes(), которые похожи на аналогично именованные методы в строках. Кроме того, есть также методы sort() и reverse(), предназначенные для переупорядочения элементов массива. Все эти методы будут описаны в последующих подразделах.

indexOf() и lastIndexOf()

Методы indexOf() и lastIndexOf() ищут в массиве элемент с указанным значением и возвращают индекс первого найденного такого элемента или -1, если ничего не было найдено. Метод indexOf() производит поиск в массиве с начала до конца, а метод lastIndexOf() — с конца до начала:

```
let a = [0,1,2,1,0];
a.indexOf(1)          // => 1: a[1] равно 1
a.lastIndexOf(1)      // => 3: a[3] равно 1
a.indexOf(3)          // => -1: нет элементов со значением 3
```

Методы indexOf() и lastIndexOf() сравнивают свой аргумент с элементами массива с применением эквивалента операции ===. Если вместо элементарных значений ваш массив содержит объекты, тогда эти методы проверяют, указывают ли две ссылки в точности на тот же самый объект. При желании взглянуть на фактическое содержимое объекта попробуйте взамен использовать метод find() с собственной специальной функцией предиката.

Методы `indexOf()` и `lastIndexOf()` принимают необязательный второй аргумент, указывающий индекс в массиве, с которого должен начинаться поиск. Если второй аргумент опущен, тогда `indexOf()` начинает с начала, а `lastIndexOf()` — с конца. Во втором аргументе разрешено передавать отрицательные значения, которые считаются смещениями с конца массива, как было в случае метора `slice()`: скажем, значение –1 задает последний элемент массива.

Показанная далее функция ищет в массиве указанное значение и возвращает массив *всех* индексов с совпадающими элементами. Она демонстрирует применение второго аргумента метода `indexOf()` для нахождения совпадений после первого.

```
// Ищет все вхождения значения x в массиве a и возвращает
// массив индексов с совпадающими элементами
function findall(a, x) {
   let results = [],    // Массив индексов, который будет возвращен
      len = a.length,  // Длина массива, в котором производится поиск
      pos = 0;         // Позиция, с которой начинается поиск
   while(pos < len) {  // Пока есть элементы для поиска...
      pos = a.indexOf(x, pos); // Искать
      if (pos === -1) break;    // Если ничего не найдено,
                                // тогда все закончено
      results.push(pos);        // Иначе сохранить индекс в массиве
      pos = pos + 1; //И начать дальнейший поиск со следующего элемента
   }
   return results;              // Возвратить массив индексов
}
```

Обратите внимание, что строки имеют методы `indexOf()` и `lastIndexOf()`, которые работают подобно описанным выше методам массивов, но только отрицательный второй аргумент трактуется как ноль.

includes()

Метод `includes()` из ES2016 принимает единственный аргумент и возвращает `true`, если массив содержит значение аргумента, или `false` в противном случае. Он не сообщает индекс, где находится значение, а только то, что оно существует. Метод `includes()` фактически является проверкой членства во множестве для массивов. Тем не менее, имейте в виду, что массивы не считаются эффективным представлением для множеств, и если вы работаете с немалым количеством элементов, то должны использовать настоящий объект `Set` (см. подраздел 11.1.1).

Метод `includes()` отличается от метода `indexOf()` одной важной особенностью. Метод `indexOf()` проверяет равенство с применением такого же алгоритма, как у операции `===`, и этот алгоритм проверки равенства считает значение "не число" отличным от любого другого значения, включая самого себя. Метод `includes()` использует немного отличающуюся версию алгоритма проверки равенства, которая рассматривает значение `NaN` как равное самому себе. Таким образом, `indexOf()` не будет обнаруживать значение `NaN` в массиве, но `includes()` будет:

```
let a = [1,true,3,NaN];
a.includes(true)            // => true
a.includes(2)               // => false
a.includes(NaN)             // => true
a.indexOf(NaN)              // => -1; indexOf не может отыскивать NaN
```

sort()

Метод `sort()` сортирует элементы массива на месте и возвращает отсортированный массив. При вызове без аргументов `sort()` сортирует элементы массива в алфавитном порядке (при необходимости временно преобразуя их в строки для выполнения сравнений):

```
let a = ["banana", "cherry", "apple"];
a.sort();                   // a == ["apple", "banana", "cherry"]
```

Если массив содержит неопределенные элементы, тогда в результате сортировки они попадают в конец массива.

Чтобы отсортировать массив не в алфавитном порядке, вам придется передать методу `sort()` функцию сравнения в виде аргумента. Такая функция решает, какой из двух ее аргументов должен появляться первым в отсортированном массиве. Если первый аргумент должен появляться перед вторым, то функция сравнения обязана возвращать число меньше нуля. Если первый аргумент должен появляться после второго, тогда функция сравнения обязана возвращать число больше нуля. А если два значения эквивалентны (т.е. порядок их следования неважен), то функция сравнения должна возвращать 0. Итак, например, для сортировки элементов массива в числовом, а не алфавитном порядке вы можете записать приведенный ниже код:

```
let a = [33, 4, 1111, 222];
a.sort();                   // a == [1111, 222, 33, 4]; алфавитный порядок
a.sort(function(a,b) { // Передать функцию сравнения
   return a-b; //Возвращает число < 0, 0 или > 0 в зависимости от порядка
});                         //a == [4, 33, 222, 1111]; числовой порядок
a.sort((a,b) => b-a); //a == [1111,222,33,4];обратный числовой порядок
```

В качестве еще одного примера сортировки элементов массива вы можете выполнить нечувствительную к регистру сортировку в алфавитном порядке массива строк, передав `sort()` функцию сравнения, которая приводит оба своих аргумента к нижнему регистру (с помощью метода `toLowerCase()`), прежде чем сравнивать их:

```
let a = ["ant", "Bug", "cat", "Dog"];
a.sort();                   // a == ["Bug","Dog","ant","cat"]; сортировка,
                            // чувствительная к регистру
a.sort(function(s,t) {
   let a = s.toLowerCase();
   let b = t.toLowerCase();
   if (a < b) return -1;
   if (a > b) return 1;
   return 0;
});                         // a == ["ant","Bug","cat","Dog"]; сортировка,
                            // нечувствительная к регистру
```

reverse()

Метод reverse() изменяет на противоположный порядок следования элементов в массиве и возвращает обращенный массив. Он делает это на месте; другими словами, reverse() не создает новый массив с переупорядоченными элементами, а взамен переставляет их в существующем массиве:

```
let a = [1,2,3];
a.reverse();                    // a == [3,2,1]
```

7.8.7. Преобразования массивов в строки

Класс Array определяет три метода, которые способны преобразовывать массивы в строки, что обычно может делаться при создании журнальных сообщений и сообщений об ошибках. (Если вы хотите сохранить содержимое массива в текстовой форме для дальнейшего применения, тогда вместо использования описанных здесь методов сериализируйте массив посредством JSON.stringify().)

Метод join() преобразует все элементы массива в строки и выполняет их конкатенацию, возвращая результирующую строку. Вы можете указать необязательную строку, которая разделяет элементы в результирующей строке. Если строка разделителя не задана, тогда применяется запятая:

```
let a = [1, 2, 3];
a.join()                        // => "1,2,3"
a.join(" ")                     // => "1 2 3"
a.join("")                      // => "123"
let b = new Array(10);          // Массив с длиной 10 без элементов
b.join("-")                     // => "---------": строка из 9 дефисов
```

Метод join() является противоположностью метода String.split(), который создает массив, разбивая строку на части.

Как и все объекты JavaScript, массивы метод toString(). Для массива он работает подобно методу join(), вызываемому без аргументов:

```
[1,2,3].toString()              // => "1,2,3"
["a", "b", "c"].toString()      // => "a,b,c"
[1, [2,"c"]].toString()         // => "1,2,c"
```

Обратите внимание, что вывод не включает квадратные скобки или ограничитель любого другого вида вокруг значения типа массива.

Метод toLocaleString() представляет собой локализованную версию toString(). Он преобразует каждый элемент массива в строку, вызывая метод toLocaleString() элемента, после чего выполняет конкатенацию результирующих строк с использованием специфической к локали (определенной реализацией) строки разделителя.

7.8.8. Статические функции массивов

В дополнение к уже документированным методам массивов класс Array также определяет три статических функции, которые можно вызывать через конс-

труктор Array, а не на массивах. Array.of() и Array.from() — это фабричные методы для создания новых массивов. Они рассматривались в подразделах 7.1.4 и 7.1.5.

Еще одна статическая функция массива, Array.isArray(), полезна для определения, является ли неизвестное значение массивом:

```
Array.isArray([])      // => true
Array.isArray({})      // => false
```

7.9. Объекты, похожие на массивы

Как было показано, массивы JavaScript обладают рядом специальных особенностей, не присущих другим объектам.

- Свойство length автоматически обновляется при добавлении новых элементов в массив.
- Установка length в меньшее значение усекает массив.
- Массивы наследуют полезные методы от Array.prototype.
- Array.isArray() возвращает true для массивов.

Перечисленные особенности делают массивы JavaScript отличающимися от обыкновенных объектов. Но они не являются жизненно важными средствами, которые определяют массив. Часто вполне разумно трактовать любой объект с числовым свойством length и соответствующими неотрицательными целочисленными свойствами как разновидность массива.

В действительности такие "похожие на массивы" объекты изредка встречаются на практике, и хотя вы не можете напрямую вызывать на них методы массивов или ожидать специального поведения от свойства length, проходить по ним по-прежнему удастся с помощью того же самого кода, который применялся для подлинных массивов. Оказывается, что многие алгоритмы для массивов работают так же хорошо с объектами, похожими на массивы, как с настоящими массивами. Сказанное особенно справедливо, если ваши алгоритмы обращаются с массивом как с допускающим только чтение и, по крайней мере, оставляют длину массива неизмененной.

В следующем коде берется обыкновенный объект, к нему добавляются свойства, чтобы сделать его объектом, похожим на массив, и затем выполняется итерация по "элементам" результирующего псевдомассива:

```
let a = {};  // Начать с обыкновенного пустого объекта
// Добавить свойства, чтобы сделать его "похожим на массив"
let i = 0;
while(i < 10) {
   a[i] = i * i;
   i++;
}
a.length = i;
```

```
// Выполнить итерацию по нему, как если бы он был настоящим массивом
let total = 0;
for(let j = 0; j < a.length; j++) {
   total += a[j];
}
```

В коде JavaScript на стороне клиента несколько методов для работы с HTML-документами (скажем, `document.querySelectorAll()`) возвращают объекты, похожие на массивы. Ниже показана функция, которую вы могли бы использовать для проверки, работают ли объекты подобно массивам:

```
// Определяет, является ли о объектом, похожим на массив.
// Строки и функции имеют числовые свойства length, но исключаются
// проверкой typeof. В коде JavaScript на стороне клиента текстовые
// узлы DOM имеют числовое свойство length, и может понадобиться их
// исключить с помощью дополнительной проверки o.nodeType !== 3.
function isArrayLike(o) {
   if (o &&                          // o - не null, undefined и т.д.
      typeof o === "object" &&       // o - объект
      Number.isFinite(o.length) &&   // o.length - конечное число
      o.length >= 0 &&               // o.length - неотрицательное
      Number.isInteger(o.length) &&  // o.length - целое число
      o.length < 4294967295) {       // o.length < 2^32 - 1
      return true;                   // Тогда объект о похож на массив
   } else {
      return false;                  // Иначе объект о не похож на массив
   }
}
```

В последнем разделе мы увидим, что строки ведут себя подобно массивам. И все же проверки вроде приведенной выше для объектов, похожих на массив, обычно возвращают `false` для строк — как правило, такие объекты лучше обрабатываются как строки, а не как массивы.

Большинство методов массивов JavaScript преднамеренно определены как обобщенные, чтобы помимо подлинных массивов они корректно работали и в случае применения к объектам, похожим на массивы. Поскольку объекты, похожие на массивы, не наследуются от `Array.prototype`, вам не удастся вызывать на них методы массивов напрямую. Однако вы можете вызывать их косвенно, используя метод `Function.call` (детали ищите в подразделе 8.7.4):

```
let a = {"0": "a", "1": "b", "2": "c", length: 3}; // Объект,
                                                   // похожий на массив
Array.prototype.join.call(a, "+")                  // => "a+b+c"
Array.prototype.map.call(a, x => x.toUpperCase()) // => ["A","B","C"]
Array.prototype.slice.call(a, 0)    // => ["a","b","c"]: копирование
                                    // в подлинный массив
Array.from(a)          // => ["a","b","c"]: более легкое копирование
```

Начиная со второй строки, на объекте, похожем на массив, вызывается метод `slice()` класса `Array`, чтобы скопировать элементы этого объекта в подлинный массив. Такой идиоматический трюк присутствует в большом количестве в унаследованном коде, но теперь гораздо легче задействовать `Array.from()`.

7.10. Строки как массивы

Строки JavaScript ведут себя как допускающие только чтение массивы символов Unicode в кодировке UTF-16. Вместо доступа к индивидуальным символам с помощью метода `charAt()` вы можете применять квадратные скобки:

```
let s = "test";
s.charAt(0)        // => "t"
s[1]               // => "e"
```

Разумеется, операция `typeof` по-прежнему возвращает `"string"` для строк, а метод `Array.isArray()` возвращает `false`, если ему передана строка.

Основное преимущество индексируемых строк заключается в том, что мы можем заменить вызовы `charAt()` квадратными скобками, которые более лаконичны и читабельны и потенциально более эффективны. Тем не менее, тот факт, что строки ведут себя подобно массивам, также означает возможность применения к ним обобщенных методов массивов. Например:

```
Array.prototype.join.call("JavaScript", " ") // => "J a v a S c r i p t"
```

Имейте в виду, что строки являются неизменяемыми значениями, поэтому при обхождении с ними как с массивами они будут массивами, доступными только для чтения. Методы массивов вроде `push()`, `sort()`, `reverse()` и `splice()` модифицируют массив на месте и со строками не работают. Однако попытка модифицировать строку с использованием метода массива не приводит к ошибке: она просто молча заканчивается неудачей.

7.11. Резюме

В главе была подробно раскрыта тема массивов JavaScript, включая детали о таких экзотических понятиях, как разреженные массивы и объекты, похожие на массивы. Ниже перечислены основные моменты, рассмотренные в главе, которые следует запомнить.

- Литералы типа массивов записываются как списки разделенных запятыми значений внутри квадратных скобок.
- Доступ к индивидуальным элементам массива производится путем указания желаемого индекса в массиве внутри квадратных скобок.
- Цикл `for/of` и операция распространения `...`, введенные в ES6, являются чрезвычайно удобными способами итерации по массивам.
- В классе `Array` определен богатый набор методов для манипулирования массивами, и вы обязательно должны ознакомиться с API-интерфейсом `Array`.

ГЛАВА 8

Функции

В этой главе рассматриваются функции JavaScript. Функции являются фундаментальными строительными блоками для программ на JavaScript и распространенным средством почти во всех языках программирования. Возможно, вы уже знакомы с концепцией функции, но под другим названием, таким как *подпрограмма* или *процедура*.

Функция — это блок кода JavaScript, который определяется однажды, но может выполняться, или *вызываться*, любое количество раз. Функции JavaScript *параметризированы*: определение функции может включать список идентификаторов, известных как *параметры*, которые выступают в качестве локальных переменных для тела функции. При вызове функций для параметров предоставляются значения, или *аргументы*. Функции часто используют значения своих аргументов для вычисления *возвращаемого значения*, которое становится значением выражения вызова функции. В дополнение к аргументам каждый вызов имеет еще одно значение — *контекст вызова*, который представляет собой значение ключевого слова `this`.

Если функция присваивается свойству объекта, тогда она называется методом данного объекта. Когда функция вызывается на объекте или через объект, то такой объект будет контекстом вызова или значением `this` для функции. Функции, предназначенные для инициализации вновь созданных объектов, называются конструкторами. Конструкторы были описаны в разделе 6.2, а также будут подробно обсуждаться в главе 9.

В JavaScript функции являются объектами, и программы могут манипулировать ими. Например, в JavaScript можно присваивать функции переменным и передавать их другим функциям. Поскольку функции — это объекты, то допускается устанавливать их свойства и даже вызывать их методы.

Определения функций JavaScript можно вкладывать внутрь других функций, и они имеют доступ к любым переменным, находящимся в области видимости, где они определены. Таким образом, функции JavaScript представляют собой *замыкания*, что делает возможными важные и мощные методики программирования.

8.1. Определение функций

Самый простой способ определения функции JavaScript предусматривает применение ключевого слова `function`, которое можно использовать как объявление или как выражение. В версии ES6 устанавливается важный новый способ определения функций без ключевого слова `function`: "стрелочные функции" обладают чрезвычайно компактным синтаксисом и удобны, когда одна функция передается в виде аргумента другой функции. В последующих подразделах раскрываются три способа определения функций. Имейте в виду, что некоторые детали синтаксиса определения функций, касающиеся параметров функций, отложены до раздела 8.3.

В объектных литералах и определениях классов имеется удобный сокращенный синтаксис для определения методов. Он был описан в подразделе 6.10.5 и эквивалентен присваиванию выражения определения функции свойству объекта с применением базового синтаксиса `имя:значение` объектных литералов. В еще одном особом случае в объектных литералах можно использовать ключевые слова `get` и `set` для определения специальных методов получения и установки свойств. Такой синтаксис определения функций был раскрыт в подразделе 6.10.6.

Обратите внимание, что функции также можно определять с помощью конструктора `Function()`, который обсуждается в подразделе 8.7.7. Кроме того, в JavaScript определено несколько специализированных видов функций. Скажем, `function*` определяет генераторные функции (см. главу 12), а `async function` — асинхронные функции (см. главу 13).

8.1.1. Объявления функций

Объявления функций состоят ключевого слова `function`, за которым следуют перечисленные ниже компоненты.

- Идентификатор, именующий функцию. Имя является обязательной частью объявления функции: оно служит именем переменной, которой присваивается вновь созданный объект функции.
- Пара круглых скобок вокруг списка из нуля или большего количества идентификаторов, разделенных запятыми. Идентификаторы представляют собой имена параметров функции и в теле функции ведут себя подобно локальным переменным.
- Пара фигурных скобок с нулем и более операторов JavaScript внутри. Операторы образуют тело функции: они выполняются всякий раз, когда функция вызвана.

Вот несколько примеров объявлений функций:

```
// Выводит имя и значение каждого свойства o. Возвращает undefined.
function printprops(o) {
    for(let p in o) {
        console.log(`${p}: ${o[p]}\n`);
    }
}
```

```
// Вычисляет расстояние между точками (x1,y1) и (x2,y2)
// в декартовых координатах.
function distance(x1, y1, x2, y2) {
   let dx = x2 - x1;
   let dy = y2 - y1;
   return Math.sqrt(dx*dx + dy*dy);
}
// Рекурсивная функция (та, что вызывает саму себя), которая вычисляет
// факториалы. Вспомните, что x! - это произведение x
// и всех положительных целых чисел, меньших x.
function factorial(x) {
   if (x <= 1) return 1;
   return x * factorial(x-1);
}
```

Один из важных моментов относительно объявлений функций, который нужно понимать, связан с тем, что имя функции становится переменной, чьим значением будет сама функция. Операторы объявления функций "поднимаются" в начало включающего сценария, функции или блока, поэтому функции, определяемые подобным образом, можно вызывать в коде, находящемся перед определением. По-другому можно сказать, что все функции, объявленные в блоке кода JavaScript, будут определены повсюду в этом блоке, и они определяются перед тем, как интерпретатор JavaScript начинает выполнять любой код в данном блоке.

Описанные выше функции `distance()` и `factorial()` предназначены для вычисления значения. Для возвращения результирующего значения вызывающему коду в них применяется оператор `return`, который приводит к тому, что функция останавливает выполнение, и `return` возвращает значение своего выражения (если задано) в вызывающий код. Если оператор `return` не имеет ассоциированного выражения, тогда возвращаемым значением функции будет `undefined`.

Функция `printprops()` отличается: ее работа заключается в выводе имен и значений свойств объекта. Необходимость в возвращаемом значении отсутствует, а потому в ней нет оператора `return`. Значением выражения вызова функции `printprops()` всегда будет `undefined`. Если функция не содержит оператор `return`, тогда операторы в теле функции выполняются, пока не будет достигнут конец, и вызывающему коду возвращается значение `undefined`.

До выхода ES6 объявлять функции разрешалось только на верхнем уровне внутри файла JavaScript или внутри другой функции. Хотя некоторые реализации нарушали это правило, формально не допускалось определять функции внутри тела циклов, условных операторов или других блоков. Однако в строгом режиме ES6 объявления функций разрешено помещать внутрь блоков. Тем не менее, функция, определенная внутри блока, существует только внутри этого блока и снаружи него будет невидимой.

8.1.2. Выражения функций

Выражения функций выглядят во многом похоже на объявления функций, но они встречаются внутри контекста более крупного выражения или оператора и наличие имени у них необязательно. Ниже приведены примеры выражений функций:

```
// Это выражение функции определяет функцию, которая возводит
// свой аргумент в квадрат.
// Обратите внимание, что мы присваиваем ее переменной.
const square = function(x) { return x*x; };

// Выражения функций могут включать имена, которые полезны для рекурсии.
const f = function fact(x) { if (x <= 1) return 1;
                             else return x*fact(x-1); };

// Выражения функций могут также использоваться как аргументы
// других функций:
[3,2,1].sort(function(a,b) { return a-b; });

// Выражения функций иногда определяются и немедленно вызываются:
let tensquared = (function(x) {return x*x;}(10));
```

Обратите внимание, что для функций, определяемых как выражения, имя функции необязательно, и в большинстве предшествующих выражений функций оно опущено. Объявление функции на самом деле *объявляет* переменную и присваивает ей объект функции. С другой стороны, выражение функции не объявляет переменную: вы можете присвоить вновь определенный объект функции константе или переменной, если собираетесь многократно на него ссылаться. С выражениями функций рекомендуется использовать `const`, чтобы непреднамеренно не переписать свои функции, присваивая новые значения.

Функциям разрешено назначать имя, как в функции вычисления факториалов, которой необходимо ссылаться на саму себя. Если выражение функции содержит имя, тогда локальная область видимости такой функции будет включать привязку этого имени к объекту функции. В действительности имя функции становится локальной переменной внутри функции. Большинству функций, которые определены как выражения, имена не нужны, что делает их определение более компактным (хотя и не настолько компактным как стрелочные функции, рассматриваемые далее).

Существует важное отличие между определением функции `f()` с помощью объявления функции и присваиванием функции переменной `f` после ее создания посредством выражения. Когда применяется форма объявления, объекты функций создаются до того, как содержащий их код начнет выполняться, а определения поднимаются, чтобы функции можно было вызывать в коде, который находится выше оператора определения. Однако это не так в случае функций, определенных как выражения: такие функции не существуют до тех пор, пока не будет фактически вычислено выражение, которое их определяет. Кроме того, для того, чтобы вызвать функцию, вы должны быть в состоянии ссылаться на нее, а сослаться на функцию, определенную как выражение, не удастся до тех пор, пока она не будет присвоена переменной, поэтому функции, определяемые с помощью выражений, нельзя вызывать до их определения.

8.1.3. Стрелочные функции

В ES6 функции можно определять с использованием чрезвычайно компактного синтаксиса, который называется "стрелочными функциями". Этот синтаксис напоминает математическую запись и применяет "стрелку" => для отделения параметров функции от ее тела. Ключевое слово `function` не используется и поскольку стрелочные функции являются выражениями, а не операторами, то также нет необходимости в имени функции. Общая форма стрелочной функции выглядит как список разделенных запятыми параметров в круглых скобках, за которым следует стрелка => и затем тело функции в фигурных скобках:

```
const sum = (x, y) => { return x + y; };
```

Но стрелочные функции поддерживают даже более компактный синтаксис. Если тело функции представляет собой одиночный оператор `return`, тогда можно опустить ключевое слово `return`, сопровождающую его точку с запятой и фигурные скобки и записать тело функции в виде выражения, значение которого возвращается:

```
const sum = (x, y) => x + y;
```

Кроме того, если стрелочная функция имеет в точности один параметр, то круглые скобки вокруг списка параметров можно опустить:

```
const polynomial = x => x*x + 2*x + 3;
```

Тем не менее, важно запомнить, что стрелочная функция без параметров должна быть записана с пустой парой круглых скобок:

```
const constantFunc = () => 42;
```

Обратите внимание, что при написании стрелочной функции не следует помещать символ новой строки между параметрами функции и стрелкой =>. В противном случае получится строка вроде `const polynomial = x`, которая сама по себе является допустимым оператором присваивания.

Вдобавок, если тело стрелочной функции — это одиночный оператор `return`, но возвращаемым выражением оказывается объектный литерал, тогда его придется поместить внутрь круглых скобок во избежание синтаксической неоднозначности между фигурными скобками тела функции и фигурными скобками объектного литерала:

```
const f = x => { return { value: x }; };  // Правильно: f()
                                          // возвращает объект
const g = x => ({ value: x });  // Правильно: g() возвращает объект
const h = x => { value: x };    // Неправильно: h() ничего не возвращает
const i = x => { v: x, w: x }; // Неправильно: синтаксическая ошибка
```

В третьей строке кода функция `h()` по-настоящему неоднозначна: код, который был задуман в качестве объектного литерала, может быть разобран как помеченный оператор, поэтому создается функция, возвращающая `undefined`. Однако в четвертой строке более сложный объектный литерал не является допустимым оператором, и такой незаконный код становится причиной синтаксической ошибки.

Лаконичный синтаксис стрелочных функций делает их идеальными, когда нужно передавать одну функцию в другую, как часто приходится поступать с методами массивов наподобие `map()`, `filter()` и `reduce()` (см. подраздел 7.8.1), например:

```
// Создать копию массива, не содержащую элементы null:
let filtered = [1,null,2,3].filter(x => x !== null);  // filtered ==
                                                      // [1,2,3].
// Возвести в квадрат ряд чисел:
let squares = [1,2,3,4].map(x => x*x);   // squares == [1,4,9,16]
```

Стрелочные функции отличаются от функций, определенных другими способами, одним критически важным аспектом: они наследуют значение ключевого слова `this` из среды, в которой определены, а не определяют собственный контекст вызова, как делают функции, определяемые другими способами. Это важная и очень полезная характеристика стрелочных функций, к которой мы еще возвратимся позже в главе. Стрелочные функции также отличаются от других функций тем, то они не имеют свойства `prototype`, т.е. их нельзя применять в качестве функций конструкторов для новых классов (см. раздел 9.2).

8.1.4. Вложенные функции

В JavaScript функции могут быть вложенными внутри других функций. Например:

```
function hypotenuse(a, b) {
   function square(x) { return x*x; }
   return Math.sqrt(square(a) + square(b));
}
```

Интересной особенностью вложенных функций является их правила видимости переменных: они могут получать доступ к параметрам и переменным функции (или функций), внутрь которой вложены. Скажем, в показанном выше коде внутренняя функция `square()` может читать и записывать параметры `a` и `b`, определенные во внешней функции `hypotenuse()`. Такие правила областей видимости для вложенных функций очень важны и мы снова обратимся к ним в разделе 8.6.

8.2. Вызов функций

Код JavaScript, который образует тело функции, выполняется не при определении функции, а когда она вызвана. Функции JavaScript могут вызываться пятью способами:

- как функции;
- как методы;
- как конструкторы;
- косвенно посредством их методов `call()` и `apply()`;
- неявно через языковые средства JavaScript, которые не выглядят похожими на нормальные вызовы функций.

8.2.1. Вызов функции

Функции вызываются как функции или как методы с помощью выражения вызова (см. раздел 4.5). Выражение вызова состоит из выражения функции, вычисляемого в объект функции, за которым следует открывающая круглая скобка, список из нуля и более выражений аргументов, разделяемых запятыми, и закрывающая круглая скобка. Если выражение функции представляет собой выражение доступа к свойству, т.е. функция является свойством объекта или элементом массива, тогда это выражение вызова метода. Такой случай объясняется в следующем разделе. В показанном ниже коде содержится несколько обыкновенных выражений вызова функций:

```
printprops({x: 1});
let total = distance(0,0,2,1) + distance(2,1,3,5);
let probability = factorial(5)/factorial(13);
```

В вызове каждое выражение (между круглыми скобками) вычисляется, а результирующие значения становятся аргументами функции. Эти значения присваиваются параметрам, имена которых указаны в определении функции. В теле функции ссылка на параметр вычисляется как значение соответствующего аргумента.

Для обыкновенных вызовов функций возвращаемое значение функции становится значением выражения вызова. Если возврат из функции происходит из-за того, что интерпретатор достиг конца, то возвращаемым значением будет `undefined`. Если возврат из функции происходит из-за того, что интерпретатор выполняет оператор `return`, то возвращаемым значением будет значение выражения, следующего за `return`, или `undefined` при отсутствии значения в `return`.

Условный вызов

Версия ES2020 позволяет вставлять `?.` после выражения функции и перед открывающей круглой скобкой в вызове функции, чтобы вызывать функцию, только если она не `null` или `undefined`. То есть выражение `f?.(x)` эквивалентно (при условии отсутствия побочных эффектов) такому выражению:

```
(f !== null && f !== undefined) ? f(x) : undefined
```

Синтаксис условного вызова рассматривался в подразделе 4.5.1.

Для вызова функций в нестрогом режиме контекстом вызова (значение `this`) будет глобальный объект. Тем не менее, в строгом режиме контекстом вызова является `undefined`. Обратите внимание, что функции, определенные с использованием стрелочного синтаксиса, ведут себя по-другому: они всегда наследуют значение `this`, которое действует там, где они определены.

В функциях, написанных для вызова как функций (не как методов), обычно вообще не применяется ключевое слово `this`. Однако `this` можно использовать для определения, действует ли строгий режим:

```
// Определение и вызов функции для выяснения,
// находимся ли мы в строгом режиме
const strict = (function() { return !this; }());
```

Рекурсивные функции и стек

Рекурсивная функция — это функция, подобная `factorial()` в начале главы, которая вызывает саму себя. Некоторые алгоритмы, например, работающие с древовидными структурами данных, могут быть особенно элегантно реализованы с помощью рекурсивных функций. Тем не менее, при написании рекурсивной функции важно учитывать ограничения, связанные с памятью. Когда функция A вызывает функцию B, после чего функция B вызывает функцию C, интерпретатор JavaScript должен отслеживать контексты выполнения для всех трех функций. Когда функция C завершается, интерпретатору необходимо знать, где возобновить выполнение функции B, а когда заканчивает работу функция B, то ему нужно знать, где возобновить выполнение функции A. Вы можете представлять себе такие контексты выполнения как стек. Когда одна функция вызывает другую функцию, новый контекст выполнения помещается в стек. После возврата из этой другой функции объект контекста выполнения первой функции извлекается из стека. Если функция вызывает саму себя рекурсивно 100 раз, тогда в стек будут помещены 100 объектов и затем те же 100 объектов извлечены. Такой стек вызовов занимает память. На современном оборудовании обычно нормально писать рекурсивные функции, которые вызывают сами себя сотни раз. Но если функция вызывает саму себя десять тысяч раз, то вполне вероятно, что она потерпит неудачу с выдачей сообщения об ошибке вроде `Maximum call-stack size exceeded` (Превышен максимальный размер стека вызовов).

8.2.2. Вызов метода

Метод — это не более чем функция JavaScript, которая хранится в свойстве объекта. Имея функцию `f` и объект `o`, определить метод по имени `m` объекта `o` можно следующим образом:

```
o.m = f;
```

После определения метода `m()` в объекте `o` его можно вызвать:

```
o.m();
```

Или, если метод `m()` ожидает передачи двух аргументов, тогда он вызывается так:

```
o.m(x, y);
```

Приведенный выше код является выражением вызова: оно включает выражение функции `o.m` и два выражения аргументов, `x` и `y`. Само выражение функции представляет собой выражение доступа к свойству, а это означает, что функция вызывается как метод, а не обыкновенная функция.

Аргументы и возвращаемое значение вызова метода обрабатываются точно так же, как было описано для вызова обычной функции. Однако вызовы методов отличаются от вызовов функций одним важным аспектом: контекстом вызова. Выражения доступа к свойствам состоят из двух частей: объекта (в данном случае `o`) и имени свойства (`m`). В выражении вызова метода подобного рода объект `o` становится контекстом вызова и в теле функции можно ссылаться на этот объект с применением ключевого слова `this`. Ниже показан конкретный пример:

```
let calculator = {  // Объектный литерал
    operand1: 1,
    operand2: 1,
    add() {     // Для этой функции мы используем сокращенный
                // синтаксис методов
        // Обратите внимание на применение ключевого слова this
        // для ссылки на вмещающий объект.
        this.result = this.operand1 + this.operand2;
    }
};
calculator.add();   // Вызов метода для вычисления 1+1
calculator.result   // => 2
```

В большинстве вызовов методов для доступа к свойствам используется точечная запись, но выражения доступа к свойствам, в которых применяются квадратные скобки, тоже приводят к вызову методов. Например, следующие строки кода представляют собой вызовы методов:

```
o["m"](x,y); // Еще один способ записи o.m(x,y)
a[0](z)      // Также вызов метода (Предполагается, что a[0] - функция)
```

Вызовы методов могут также включать в себя более сложные выражения доступа к свойствам:

```
customer.surname.toUpperCase();  // Вызов метода на customer.surname
f().m();     // Вызов метода m() на возвращаемом значении f()
```

Методы и ключевое слово `this` занимают центральное место в парадигме объектно-ориентированного программирования. Любой функции, которая используется как метод, фактически передается неявный аргумент — объект, через который она вызывается. Обычно метод выполняет действие определенного вида над этим объектом и синтаксис вызова методов оказывается элегантным способом выражения того факта, что функция оперирует с объектом. Сравните следующие две строки:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

Гипотетические функции, вызываемые в приведенных двух строках кода, могут выполнять то же самое действие над (гипотетическим) объектом `rect`, но синтаксис вызова метода в первой строке более четко отражает идею о том, что главной целью операции является объект `rect`.

Формирование цепочек методов

Когда методы возвращают объекты, возвращаемое значение одного вызова метода можно применять как часть дальнейшего вызова. В результате получается последовательность (или "цепочка") вызовов методов в форме одиночного выражения. Скажем, при работе с асинхронными операциями, основанными на `Promise` (см. главу 13), часто приходится писать код, структурированный примерно так:

```
//Последовательно запустить три асинхронных операции, обрабатывая ошибки
doStepOne().then(doStepTwo).then(doStepThree).catch(handleErrors);
```

Когда вы пишете метод, который не имеет собственного возвращаемого значения, подумайте о том, чтобы метод возвращал `this`. Если вы будете поступать так согласованно повсюду в своем API-интерфейсе, то получите возможность использовать стиль программирования, известный как *формирование цепочек методов*[1], при котором объект указывается один раз и затем на нем можно вызывать множество методов:

```
new Square().x(100).y(100).size(50).outline("red").fill("blue").draw();
```

Обратите внимание, что `this` — это ключевое слово, не имя переменной или свойства. Синтаксис JavaScript не разрешает присваивать значение ключевому слову `this`.

Область видимости ключевого слова `this` не ограничивается как у переменных, и за исключением стрелочных функций вложенные функции не наследуют значение `this` вмещающей функции. Если вложенная функция вызывается как метод, то ее значение `this` будет объектом, на котором она вызывалась. Если вложенная функция (т.е. не стрелочная функция) вызывается как функция, тогда ее значением `this` будет или глобальный объект (в нестрогом режиме), или `undefined` (в строгом режиме). Предположение о том, что вложенная функция, определенная внутри метода и вызываемая как функция, может применять `this` для получения контекста вызова метода, является распространенной ошибкой. Проблема демонстрируется в следующем коде:

```
let o = {                 // Объект o
   m: function() {        // Метод m объекта
      let self = this;    // Сохранить значение this в переменной
      this === o          // => true: this является объектом o
      f();                // Теперь вызвать вспомогательную функцию f()

      function f() {      // Вложенная функция f
         this === o       // => false: this является глобальным объектом
                          // или undefined
         self === o       // => true: self является внешним значением this
      }
   }
};
o.m();                    // Вызвать метод m на объекте o
```

[1] Термин придумал Мартин Фаулер. См. http://martinfowler.com/dslCatalog/methodChaining.html.

Внутри вложенной функции f() ключевое слово this не равно объекту o. По широкому признанию это считается изъяном языка JavaScript, и о нем важно знать. Выше в коде показан один распространенный обходной путь. Внутри метода m мы присваиваем значение this переменной self и внутри вложенной функции f для ссылки на вмещающий объект мы можем использовать self вместо this.

В ES6 и последующих версиях еще один способ обхода проблемы предусматривает преобразование вложенной функции f в стрелочную функцию, которая надлежащим образом наследует значение this:

```
const f = () => {
    this === o  // true, поскольку стрелочные функции наследуют this
};
```

Функции, определенные с помощью выражений, а не операторов, не поднимаются, поэтому для того, чтобы код заработал, определение функции f необходимо переместить внутри метода m, поместив его перед вызовом f.

Другой обходной путь заключается в вызове метода bind() вложенной функции с целью определения новой функции, которая неявно вызывается на указанном объекте:

```
const f = (function() {
  this === o //true, поскольку мы привязали эту функцию к внешнему this
}).bind(this);
```

Мы обсудим bind() в подразделе 8.7.5.

8.2.3. Вызов конструктора

Если вызов функции или метода предваряется ключевым словом new, тогда это будет вызов конструктора. (Вызовы конструкторов были представлены в разделе 4.6 и подразделе 6.2.2, а конструкторы будут более подробно раскрыты в главе 9.) Вызовы конструкторов отличаются от вызовов обыкновенных функций и методов своей обработкой аргументов, контекста вызова и возвращаемого значения.

Если вызов конструктора содержит список аргументов в круглых скобках, то выражения аргументов вычисляются и результаты передаются функции таким же способом, как для вызовов функций и методов. Хотя это не общепринятая практика, но вы можете опускать пару пустых круглых скобок в вызове конструктора. Например, следующие две строки кода эквивалентны:

```
o = new Object();
o = new Object;
```

Вызов конструктора создает новый пустой объект, который унаследован от объекта, заданного свойством prototype конструктора. Функции конструкторов предназначены для инициализации объектов, и вновь созданный объект применяется в качестве контекста вызова, поэтому функция конструктора может ссылаться на него посредством ключевого слова this. Обратите внимание, что новый объект используется как контекст вызова, даже если вызов конструк-

тора выглядит похожим на вызов метода. То есть в выражении `new o.m()` объект `o` не применяется в качестве контекста вызова.

Функции конструкторов не используют ключевое слово `return` нормальным образом. Они инициализируют новый объект и затем неявно производят возврат, когда достигают конца своего тела. В данном случае новым объектом является значение выражения вызова конструктора. Тем не менее, если в конструкторе явно применяется оператор `return` для возвращения объекта, тогда этот объект становится значением выражения вызова. Если в конструкторе используется `return` без значения или возвращается элементарное значение, то возвращаемое значение игнорируется и в качестве значения выражения вызова применяется новый объект.

8.2.4. Косвенный вызов функции

Функции JavaScript представляют собой объекты и подобно всем объектам JavaScript имеют методы. Два из них, `call()` и `apply()`, вызывают функцию косвенно. Оба метода позволяют явно указывать значение `this` для вызова, т.е. вы можете вызвать любую функцию как метод любого объекта, даже если она в действительности не является методом этого объекта. Оба метода также позволяют указывать аргументы для вызова. Метод `call()` использует в качестве аргументов для функции собственный список аргументов, а метод `apply()` ожидает массива значений, которые должны применяться как аргументы. Методы `call()` и `apply()` подробно описаны в подразделе 8.7.4.

8.2.5. Неявный вызов функции

Существуют разнообразные языковые средства JavaScript, которые не похожи на вызовы функций, но служат причиной их вызова. Проявляйте особую осторожность, когда пишете функции, которые могут быть вызваны неявно, поскольку ошибки, побочные эффекты и проблемы с производительностью в таких функциях труднее диагностировать и исправлять, чем в обыкновенных функциях, по той простой причине, что при инспектировании кода факты их вызова могут быть неочевидными.

Ниже перечислены языковые средства, которые способны приводить к неявному вызову функций.

- Если для объекта определены методы получения или установки, тогда запрашивание и установка значения его свойств могут вызывать упомянутые методы. Дополнительные сведения ищите в подразделе 6.10.6.
- Когда объект используется в строковом контексте (скажем, при конкатенации со строкой), вызывается его метод `toString()`. Аналогично, когда объект применяется в числовом контексте, вызывается его метод `valueOf()`. Детали ищите в подразделе 3.9.3.
- Когда вы проходите в цикле по элементам итерируемого объекта, может произойти несколько вызовов методов. В главе 12 объясняется, каким образом итераторы работают на уровне вызова функций, и демонстрирует-

ся, как писать такие методы, чтобы определять собственные итерируемые типы.

- Литерал тегированного шаблона представляет собой скрытый вызов функции. В разделе 14.5 будет показано, как писать функции, которые могут использоваться в сочетании со строками литералов шаблонов.
- Поведение прокси-объектов (описанных в разделе 14.7) полностью управляется функциями. Практически любое действие с одним из таких объектов станет причиной вызова какой-то функции.

8.3. Аргументы и параметры функций

Определения функций JavaScript не задают ожидаемый тип параметров функции, а вызовы функций не предпринимают никаких проверок типов в отношении передаваемых значений аргументов. На самом деле вызовы функций JavaScript даже не проверяют количество передаваемых аргументов. В последующих подразделах будет показано, что происходит, когда функция вызывается с меньшим количеством аргументов, чем объявленных параметров. Там также демонстрируется, каким образом можно явно проверять типы аргументов функции, если вам нужно гарантировать, что функция не будет вызываться с неподходящими аргументами.

8.3.1. Необязательные параметры и стандартные значения

Когда функция вызывается с меньшим количеством аргументов, чем объявленных параметров, оставшиеся параметры устанавливаются в свои стандартные значения, которыми обычно будут undefined. Часто полезно писать функции так, чтобы некоторые аргументы были необязательными. Вот пример:

```
// Присоединяет имена перечислимых свойств объекта o к массиву a
// и возвращает a.
// Если аргумент a опущен, тогда создает и возвращает новый массив.
function getPropertyNames(o, a) {
    if (a === undefined) a = [];  // Если undefined, то использовать
                                  // новый массив
    for(let property in o) a.push(property);
    return a;
}

// getPropertyNames() может вызываться с одним или двумя аргументами:
let o = {x: 1}, p = {y: 2, z: 3}; // Два объекта для тестирования
let a = getPropertyNames(o);      // a == ["x"]; получить свойства o
                                  // в новом массиве
getPropertyNames(p, a);           // a == ["x","y","z"]; добавить
                                  // к нему свойства p
```

Вместо применения оператора if в первой строке кода функции вы можете использовать операцию || следующим идиоматическим способом:

```
a = a || [];
```

Вспомните из подраздела 4.10.2, что операция || возвращает свой первый операнд, если он истинный, и второй операнд в противном случае. В нашей ситуации, если во втором аргументе передается любой объект, тогда функция будет применять этот объект. Но если второй аргумент опущен (или передается null либо другое ложное значение), то взамен будет использоваться пустой массив.

Имейте в виду, что при проектировании функций с необязательными параметрами вы обязаны помещать необязательные параметры в конец списка аргументов, чтобы их можно было опускать. Программист, вызывающий вашу функцию, не может опустить первый аргумент и передать второй: ему придется явно передать undefined в качестве первого аргумента.

В ES6 и последующих версиях вы можете определять стандартное значение для каждого параметра прямо в списке параметров функции. Просто поместите после имени параметра знак равенства и стандартное значение для применения в случае, если для этого параметра аргумент не предоставлен:

```
// Присоединяет имена перечислимых свойств объекта o к массиву a
// и возвращает a.
// Если аргумент a опущен, тогда создает и возвращает новый массив.
function getPropertyNames(o, a = []) {
    for(let property in o) a.push(property);
    return a;
}
```

Стандартные выражения параметров вычисляются, когда ваша функция вызывается, а не когда она определяется, так что при каждом вызове функции getPropertyNames() с одним аргументом создается и передается новый пустой массив[2]. Вероятно, рассуждать о функциях легче всего, если стандартные значения параметров являются константами (или литеральными выражениями вроде [] и {}). Но это не обязательно: например, для вычисления стандартного значения параметра вы можете использовать переменные или вызовы функций. Один интересный случай касается функций с множеством параметров: значение предыдущего параметра можно применять при определении стандартных значений параметров, которые следуют за ним:

```
// Эта функция возвращает объект, представляющий размеры прямоугольника.
// Если предоставляется только ширина, тогда сделать его высоту вдвое
// больше, чем ширину.
const rectangle = (width, height=width*2) => ({width, height});
rectangle(1)  // => { width: 1, height: 2 }
```

Приведенный код иллюстрирует, что стандартные значения параметров работают со стрелочными функциями. То же самое справедливо для сокращенных определений методов и всех остальных форм определения функций.

[2] Если вы знакомы с языком Python, тогда имейте в виду, что это отличается от Python, где каждый вызов использует одно и то же стандартное значение.

8.3.2. Параметры остатка и списки аргументов переменной длины

Стандартные значения параметров позволяют писать функции, которые можно вызывать с количеством аргументов меньшим, чем количество параметров. *Параметры остатка* делают возможным противоположный случай: они позволяют писать функции, которые можно вызывать с произвольно большим количеством аргументов, чем параметров. Ниже представлен пример функции, которая ожидает один и более числовых аргументов и возвращает наибольший из них:

```
function max(first=-Infinity, ...rest) {
   let maxValue = first;    // Начать с предположения, что первый
                            // аргумент самый большой
   // Затем пройти в цикле по оставшимся аргументам
   // в поисках наибольшего из них
   for(let n of rest) {
      if (n > maxValue) {
         maxValue = n;
      }
   }
   // Возвратить наибольший аргумент
   return maxValue;
}
max(1, 10, 100, 2, 3, 1000, 4, 5, 6)  // => 1000
```

Параметр остатка предварен тремя точками, и он должен быть последним параметром в объявлении функции. При вызове функции с параметром остатка переданные аргументы сначала присваиваются параметрам, отличным от остатка, после чего все оставшиеся аргументы сохраняются в массиве, который становится значением параметра остатка. Последний момент очень важен: внутри тела функции значением параметра остатка всегда будет массив. Массив может быть пустым, но параметр остатка никогда не будет равен `undefined`. (Из этого следует, что никогда не будет полезно — и не будет допустимо — определять стандартное значение для параметра остатка.)

Функции наподобие показанной в предыдущем примере, которые способны принимать любое количество аргументов, называются *функциями с переменным числом параметров*, *функциями с переменной арностью* или *vararg-функциями*. В книге используется наиболее разговорный термин, *vararg-функции*, который относится к ранним дням языка программирования C.

Не путайте три точки, определяющие параметр остатка в определении функции, и описываемую в подразделе 8.3.4 операцию распространения `...`, которая может применяться в вызовах функций.

8.3.3. Объект `Arguments`

Параметры остатка были введены в JavaScript в версии ES6. До выхода ES6 vararg-функции записывались с использованием объекта `Arguments`: идентифи-

катор arguments внутри тела любой функции ссылается на объект Arguments для данного вызова. Объект Arguments представляет собой объект, похожий на массив (см. раздел 7.9), который позволяет извлекать значения аргументов, переданные функции, по номеру, а не по имени. Ниже приведена функция max(), переписанная с применением объекта Arguments вместо параметров остатка:

```
function max(x) {
    let maxValue = -Infinity;
    //Пройти в цикле по аргументам в поисках наибольшего и запомнить его
    for(let i = 0; i < arguments.length; i++) {
        if (arguments[i] > maxValue) maxValue = arguments[i];
    }
    // Возвратить наибольший аргумент
    return maxValue;
}
max(1, 10, 100, 2, 3, 1000, 4, 5, 6) // => 1000
```

Объект Arguments датируется ранними днями языка JavaScript и сопряжен со странным историческим багажом, который делает его неэффективным и трудным для оптимизации, особенно за пределами строгого режима. Вы по-прежнему можете сталкиваться с кодом, в котором используется объект Arguments, но должны избегать его применения при написании любого нового кода. Если при рефакторинге старого кода вы встречаете функцию, использующую идентификатор arguments, то часто можете заменить его параметром остатка ...args. Отчасти неудачливое наследие объекта Arguments связано с тем, что в строгом режиме arguments интерпретируется как зарезервированное слово, поэтому вы не можете объявить параметр функции или локальную переменную с таким именем.

8.3.4. Операция распространения для вызовов функций

Операция распространения ... применяется для распаковки, или "распространения", элементов массива (или любого другого итерируемого объекта, такого как строки) в контексте, где ожидаются индивидуальные значения. В подразделе 7.1.2 операция распространения использовалась с литералами типа массивов. Тем же самым способом ее можно применять в вызовах функций:

```
let numbers = [5, 2, 10, -1, 9, 100, 1];
Math.min(...numbers)  // => -1
```

Обратите внимание, что ... — не настоящая операция в том смысле, что ее нельзя вычислить, чтобы получить значение. Взамен она является специальным синтаксисом JavaScript, который может использоваться в литералах типа массивов и вызовах функций.

Когда мы применяем тот же синтаксис ... в определении функции, а не в вызове, он дает эффект, противоположный операции распространения. Как объяснялось в подразделе 8.3.2, использование ... в определении функции собирает множество аргументов функции в массив. Параметры остатка и операцию распространения часто удобно применять вместе, как в следующей функции, которая принимает аргумент типа функции и возвращает инструментированную версию функции для тестирования:

```
// Эта функция принимает функцию и возвращает версию в виде оболочки
function timed(f) {
   return function(...args) {   // Собрать аргументы в массив
                                // параметра остатка
      console.log(`Вход в функцию ${f.name}`);
      let startTime = Date.now();
      try {
         // Передать все аргументы в функцию-оболочку
         return f(...args);  // Распространить args
      }
      finally {
         // Перед возвратом возвращаемого значения оболочки
         // вывести затраченное время
         console.log(`Выход из ${f.name} спустя ${Date.now()-
startTime}мс`);
      }
   };
}
// Рассчитать сумму чисел между 1 и n методом грубой силы
function benchmark(n) {
   let sum = 0;
   for(let i = 1; i <= n; i++) sum += i;
   return sum;
}
// Вызвать хронометрирующую версию тестовой функции
timed(benchmark)(1000000)  // => 500000500000; это сумма чисел
```

8.3.5. Деструктуризация аргументов функции в параметры

Когда вы вызываете функцию со списком значений аргументов, в конечном итоге эти значения присваиваются параметрам, объявленным в определении функции. Такая начальная стадия вызова функции во многом похожа на присваивание переменных. Следовательно, не должен удивлять тот факт, что мы можем использовать с функциями методики деструктурирующего присваивания (см. подраздел 3.10.3).

Если вы определяете функцию, которая имеет имена параметров внутри квадратных скобок, то сообщаете о том, что функция ожидает передачи значения типа массива для каждой пары квадратных скобок. Как часть процесса вызова аргументы типа массивов могут быть распакованы в индивидуально именованные параметры. В качестве примера предположим, что мы представляем двумерные векторы в виде массивов из двух чисел, где первый элемент — координата X, а второй — координата Y. С такой простой структурой данных мы могли бы написать следующую функцию для сложения двух векторов:

```
function vectorAdd(v1, v2) {
   return [v1[0] + v2[0], v1[1] + v2[1]];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

Код будет легче понять, если мы деструктурируем два аргумента типа векторов в более ясно именованные параметры:

```
function vectorAdd([x1,y1], [x2,y2]) {  // Распаковать 2 аргумента
                                        // в 4 параметра
   return [x1 + x2, y1 + y2];
}
vectorAdd([1,2], [3,4])  // => [4,6]
```

Аналогично, если мы определяем функцию, которая ожидает объектный аргумент, то можем деструктурировать параметры такого объекта. Давайте снова возьмем пример с вектором, но на этот раз предположим, что мы представляем векторы как объекты с параметрами x и y:

```
// Умножить вектор {x,y} на скалярное значение
function vectorMultiply({x, y}, scalar) {
   return { x: x*scalar, y: y*scalar };
}
vectorMultiply({x: 1, y: 2}, 2)  // => {x: 2, y: 4}
```

Приведенный пример деструктуризации одиночного объектного аргумента в два параметра достаточно понятен, т.к. мы применяли имена параметров, совпадающие с именами свойства входного объекта. Синтаксис будет более многословным и запутанным, когда нам необходимо деструктурировать свойства, имена которых отличаются от имен параметров. Ниже показан пример функции сложения векторов, реализованной для векторов на основе объектов:

```
function vectorAdd(
   {x: x1, y: y1},  // Распаковать 1-й объект в параметры x1 и y1
   {x: x2, y: y2}   // Распаковать 2-й объект в параметры x2 и y2
)
{
   return { x: x1 + x2, y: y1 + y2 };
}
vectorAdd({x: 1, y: 2}, {x: 3, y: 4})  // => {x: 4, y: 6}
```

Хитрость деструктурирующего синтаксиса вроде `{x:x1, y:y1}` в том, что нужно помнить, какие имена относятся к свойствам, а какие к параметрам. Правило, которое следует иметь в виду относительно деструктурирующего присваивания и деструктурирующих вызовов, заключается в том, что объявляемые переменные или параметры находятся в тех местах, где вы ожидали бы значения в объектном литерале. Таким образом, имена свойств всегда располагаются в левой стороне от двоеточия, а имена параметров (или переменных) — в правой стороне.

С деструктурирующими параметрами можно определять стандартные значения. Далее приведена функция умножения векторов, которая работает с двумерными или трехмерными векторами:

```
// Умножить вектор {x,y} или {x,y,z} на скалярное значение
function vectorMultiply({x, y, z=0}, scalar) {
   return { x: x*scalar, y: y*scalar, z: z*scalar };
}
vectorMultiply({x: 1, y: 2}, 2)  // => {x: 2, y: 4, z: 0}
```

Определенные языки (наподобие Python) разрешают коду вызывать функцию с аргументами, указанными в форме имя=значение, которая удобна при наличии многих необязательных аргументов или в случае, когда список параметров достаточно длинный и запомнить корректный порядок их следования нелегко. JavaScript не допускает этого напрямую, но вы можете приблизиться к такому стилю, деструктурируя объектный аргумент в параметры функции. Рассмотрим функцию, которая копирует заданное количество элементов из одного массива в другой с необязательно указываемыми начальными смещениями для каждого массива. Поскольку существует пять возможных параметров, часть которых имеют стандартные значения, и в вызывающем коде трудно запомнить, в каком порядке передавать аргументы, мы можем определить и вызывать функцию `arraycopy()` примерно так:

```
function arraycopy({from, to=from, n=from.length,
                    fromIndex=0, toIndex=0}) {
   let valuesToCopy = from.slice(fromIndex, fromIndex + n);
   to.splice(toIndex, 0, ...valuesToCopy);
   return to;
}
let a = [1,2,3,4,5], b = [9,8,7,6,5];
arraycopy({from: a, n: 3, to: b, toIndex: 4})  // => [9,8,7,6,1,2,3,5]
```

При деструктуризации массива вы можете определить параметр остатка для добавочных значений внутри массива, который распаковывается. Такой параметр остатка внутри квадратных скобок полностью отличается от параметра остатка для функции:

```
// Эта функция ожидает аргумент типа массива.
// Первые два элемента массива распаковываются в параметры x и y.
// Любые оставшиеся элементы сохраняются в массиве coords.
// Любые аргументы после первого массива упаковываются в массив rest.
function f([x, y, ...coords], ...rest) {
   return [x+y, ...rest, ...coords];
          // Примечание: здесь используется операция распространения
}
f([1, 2, 3, 4], 5, 6)  // => [3, 5, 6, 3, 4]
```

В ES2018 параметр остатка можно также использовать при деструктуризации объекта. Значением параметра остатка будет объект с любыми свойствами, которые не были деструктурированы. Объектные параметры остатка часто удобно применять с объектной операцией распространения, что тоже является нововведением ES2018:

```
// Умножить вектор {x,y} или {x,y,z} на скалярное значение,
// предохранив остальные свойства
function vectorMultiply({x, y, z=0, ...props}, scalar) {
   return { x: x*scalar, y: y*scalar, z: z*scalar, ...props };
}
vectorMultiply({x: 1, y: 2, w: -1}, 2)  // => {x: 2, y: 4, z: 0, w: -1}
```

Наконец, имейте в виду, что в дополнение к деструктуризации объектов и массивов аргументов вы также можете деструктурировать массивы объектов, объекты, которые имеют свойства типа массивов, и объекты, имеющие объектные свойства, практически на любую глубину. Возьмем код для работы с графикой, который представляет круги как объекты со свойствами `x`, `y`, `radius` и `color`, где свойство `color` является массивом цветовых компонентов красного, зеленого и синего. Вы могли бы определить функцию, которая ожидает передачи одиночного объекта круга, но деструктурирует его в шесть отдельных параметров:

```
function drawCircle({x, y, radius, color: [r, g, b]}) {
   // Пока не реализована
}
```

Если аргумент функции деструктурируется сложнее, чем здесь, то я считаю, что код становится не проще, а труднее для восприятия. Временами код будет гораздо понятнее, если принять более явный подход в отношении доступа к свойствам объектов и индексации массивов.

8.3.6. Типы аргументов

Параметры методов JavaScript не имеют объявленных типов, а значения, передаваемые функции, не подвергаются каким-либо проверкам типов. Вы можете сделать свой код самодокументированным за счет выбора описательных имен для параметров функций и предоставления аккуратного их описания в комментариях к каждой функции. (В качестве альтернативы почитайте в разделе 17.8 о языковом расширении, которое позволяет поместить поверх обычного языка JavaScript уровень проверки типов.)

Как было описано в разделе 3.9, интерпретатор JavaScript выполняет либеральное преобразование типов по мере необходимости. Таким образом, если вы напишете функцию, которая ожидает строковый аргумент, и затем вызовите ее со значением какого-то другого типа, то переданное значение просто преобразуется в строку, когда функция попытается использовать его как строку. Все элементарные типы могут быть преобразованы в строки и все объекты имеют методы `toString()` (даже если они не обязательно полезные), поэтому в таком случае ошибка не возникает.

Однако так получается не всегда. Снова возьмем показанный ранее метод `arraycopy()`. Он ожидает передачи одного или двух аргументов типа массивов и потерпит неудачу, если аргументы будут иметь неправильные типы. Если только вы не пишете закрытую функцию, которая будет вызываться только из близлежащих частей вашего кода, то возможно стоит добавить код для проверки типов аргументов, подобный тому, что показан ниже. Лучше, чтобы при передаче неправильных значений функция отказывала немедленно и предсказуемо, чем начинала выполнение и терпела неудачу позже с выдачей сообщения об ошибке, которое вероятно будет не особенно ясным. Вот пример функции, которая реализует проверку типов:

```
// Возвращает сумму элементов итерируемого объекта a.
// Все элементы a обязаны быть числами.
function sum(a) {
   let total = 0;
   for(let element of a) {  // Генерирует TypeError, если a
                            // не является итерируемым
      if (typeof element !== "number") {
         throw new TypeError("sum(): elements must be numbers");
                            // sum(): элементы обязаны быть числами
      }
      total += element;
   }
   return total;
}
sum([1,2,3])       // => 6
sum(1, 2, 3);      // !TypeError: 1 - не итерируемый объект
sum([1,2,"3"]);    // !TypeError: элемент 2 - не число
```

8.4. Функции как значения

Наиболее важные характерные черты функций заключаются в том, что их можно определять и вызывать. Определение и вызов функции являются синтаксическими средствами JavaScript и большинства других языков программирования. Тем не менее, функции в JavaScript — это не только синтаксис, но также значения, т.е. их можно присваивать переменным, сохранять в свойствах объектов или в элементах массивов, передавать как аргументы функциям и т.д.[3]

Чтобы понять, как функции могут быть данными JavaScript, а также синтаксисом JavaScript, рассмотрим следующее определение функции:

```
function square(x) { return x*x; }
```

Определение создает новый объект функции и присваивает его переменной `square`. На самом деле имя функции несущественно; это просто имя переменной, которая ссылается на объект функции. Функцию можно присвоить другой переменной, и она продолжит свою работу без изменений:

```
let s = square;    // Теперь s ссылается на ту же функцию, что и square
square(4)          // => 16
s(4)               // => 16
```

Помимо переменных функции также можно присваивать свойствам объектов. Как уже обсуждалось, в таком случае мы называем функции "методами":

```
let o = {square: function(x) { return x*x; }};    // Объектный литерал
let y = o.square(16);                             // y == 256
```

Когда мы присваиваем функции элементам массива, имя указывать вообще не обязательно:

```
let a = [x => x*x, 20]; // Литерал типа массива
a[0](a[1])              // => 400
```

[3] Этот аспект может показаться не особо интересным, если только вы не знакомы с более статическими языками, где функции являются частью программы, но ими нельзя манипулировать в коде.

Синтаксис в последнем примере выглядит странно, но все-таки он является законным выражением вызова функции!

В качестве примера того, насколько удобно обходиться с функциями как со значениями, взглянем на метод `Array.sort()`, который сортирует элементы массива. Поскольку существует много возможных порядков сортировки (числовой, алфавитный, по дате, по возрастанию, по убыванию и т.д.), метод `sort()` принимает в необязательном аргументе функцию, сообщающую о том, каким образом выполнять сортировку. Задача функции проста: для любых двух предоставленных значений она возвращает значение, которое указывает, какой элемент должен идти первым в отсортированном массиве. Этот аргумент типа функции делает метод `Array.sort()` полностью универсальным и безгранично гибким; он способен сортировать данные любого типа в любом мыслимом порядке. Примеры приводились в подразделе 7.8.6.

В примере 8.1 демонстрируются разновидности действий, которые можно предпринимать, когда функции применяются как значения. Пример может показаться несколько сложным, но в комментариях объясняется, что происходит.

Пример 8.1. Использование функций как данных

```
// Здесь мы определяем ряд простых функций
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Функция, которая принимает одну из определенных выше функций
// в качестве аргумента и вызывает ее с двумя операндами
function operate(operator, operand1, operand2) {
   return operator(operand1, operand2);
}

// Мы могли бы вызвать эту функцию следующим образом
// для расчета значения (2+3) + (4*5):
let i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// Для примера мы снова реализуем простые функции,
// на этот раз внутри объектного литерала
const operators = {
   add: (x,y) => x+y,
   subtract: (x,y) => x-y,
   multiply: (x,y) => x*y,
   divide: (x,y) => x/y,
   pow: Math.pow  // Работает также для предварительно определенных функций
};

// Эта функция принимает имя операции, ищет ее в объекте и затем
// вызывает с предоставленными операндами. Обратите внимание
// на синтаксис, используемый для вызова функции операции
function operate2(operation, operand1, operand2) {
   if (typeof operators[operation] === "function") {
      return operators[operation](operand1, operand2);
   }
```

```
    else throw "unknown operator";  // неизвестная операция
}
operate2("add", "hello", operate2("add", " ", "world"))  // => "hello world"
operate2("pow", 10, 2)                                   // => 100
```

8.4.1. Определение собственных свойств функций

Функции в JavaScript представляют собой не элементарные значения, а специализированный вид объекта и потому могут иметь свойства. Когда функции требуется "статическая" переменная, значение которой сохраняется между вызовами, часто удобно применять свойство самой функции. Например, предположим, что вы хотите написать функцию, которая при каждом вызове возвращает уникальное целое число. Функция никогда не должна возвращать то же самое значение дважды. Чтобы справиться с указанной задачей, функции необходимо отслеживать уже возвращенные значения, и такую информацию нужно сохранять между вызовами. Вы могли бы хранить информацию в глобальной переменной, но это излишне, потому что информация потребляется только самой функцией. Лучше хранить информацию в свойстве объекта `Function`. Ниже приведен пример функции, которая при каждом вызове возвращает уникальное целое число:

```
// Инициализировать свойство counter объекта функции.
// Объявления функций поднимаются, поэтому мы действительно
// можем делать такое присваивание до объявления функции.
uniqueInteger.counter = 0;
// При каждом вызове эта функция возвращает отличающееся целое число.
// Она использует собственное свойство для запоминания следующего
// возвращаемого значения.
function uniqueInteger() {
   return uniqueInteger.counter++; // Возвратить и инкрементировать
                                   // свойство counter
}
uniqueInteger()  // => 0
uniqueInteger()  // => 1
```

В качестве еще одного примера рассмотрим следующую функцию `factorial()`, которая использует собственные свойства (трактуя себя как массив) для кеширования ранее вычисленных результатов:

```
// Вычисляет факториалы и кеширует результаты как свойства самой функции
function factorial(n) {
  if (Number.isInteger(n) && n > 0) {//Только положительные целые числа
    if (!(n in factorial)) {//Если кешированный результат отсутствует
        factorial[n] = n * factorial(n-1); // Вычислить и
                                           // кешировать его
    }
    return factorial[n];    // Возвратить кешированный результат
  } else {
    return NaN;             // Если входные данные недопустимы
  }
}
```

```
factorial[1] = 1;  // Инициализировать кеш для хранения
                   // этого базового значения
factorial(6)       // => 720
factorial[5]       // => 120; вызов выше кеширует это значение
```

8.5. Функции как пространства имен

Переменные, объявленные внутри функции, за ее пределами не видны. По этой причине иногда полезно определять функцию, которая будет действовать просто как временное пространство имен, где можно определять переменные, не загромождая глобальное пространство имен.

Например, пусть у вас есть фрагмент кода JavaScript, который вы хотите применять в нескольких программах на JavaScript (или в случае кода JavaScript стороны клиента на нескольких веб-страницах). Предположим, что подобно большинству кода в нем определяются переменные для хранения промежуточных результатов производимых вычислений. Проблема в том, что поскольку упомянутый фрагмент кода будет использоваться во множестве программ, вы не знаете, будут ли конфликтовать создаваемые им переменные с переменными, которые создают сами программы. Решение предусматривает помещение фрагмента кода внутрь функции с последующим ее вызовом. Таким образом, переменные, которые были бы глобальными, становятся локальными по отношению к функции:

```
function chunkNamespace() {
    // Фрагмент многократно используемого кода.
    // Любые определяемые здесь переменные являются локальными
    // по отношению к этой функции и не загромождают глобальное
    // пространство имен.
}
chunkNamespace();  // Но не забудьте вызвать функцию!
```

В приведенном коде определяется единственная глобальная переменная: имя функции chunkNamespace. Если определение даже одного свойства кажется избыточным, тогда можно определить и вызвать анонимную функцию в единственном выражении:

```
(function() {      // Функция chunkNamespace(), переписанная
                   // в виде неименованного выражения
    // Фрагмент многократно используемого кода
}());               // Конец литерала типа функции и его вызов
```

Такая методика определения и вызова функции в единственном выражении применяется достаточно часто, поэтому она стала идиоматической и получила название "немедленно вызываемое выражение функции". Обратите внимание на использование круглых скобок в предыдущем примере кода. Открывающая круглая скобка перед function обязательна, потому что без нее интерпретатор JavaScript попытается разобрать ключевое слово function как оператор объявления функции. При наличии круглой скобки интерпретатор корректно распознает конструкцию как выражение определения функции. Открывающая круглая

скобка также позволяет людям понять, что функция определяется с целью немедленного вызова, а не для применения в будущем.

Подобное использование функций в качестве пространств имен становится по-настоящему полезным, когда мы определяем одну или большее количество функций внутри функции, служащей пространством имен, с применением переменных в этом пространстве имен, но затем передаем их обратно в виде возвращаемого значения функции, служащей пространством имен. Функции такого рода известны как *замыкания* и рассматриваются в следующем разделе.

8.6. Замыкания

Как и в большинстве современных языков программирования, в JavaScript используется *лексическая область видимости*. Это означает, что функции выполняются с применением области видимости переменных, которая действовала, когда они были определены, а не когда вызваны. Для реализации лексической области видимости внутреннее состояние объекта функции JavaScript должно включать не только код функции, но также ссылку на область видимости, в которой находится определение функции. Такое сочетание объекта функции и области видимости (набора привязок переменных), в которой распознаются переменные функции, в компьютерной литературе называется *замыканием*.

Формально все функции JavaScript являются замыканиями, но поскольку большинство функций вызываются из той же области видимости, где они были определены, вполне нормально не обращать внимания на вовлечение замыканий. Замыкания становятся интересными, когда они вызываются из области видимости, отличающейся от той, где они определялись. Чаще всего подобное происходит при возвращении объекта вложенной функции из функции, внутри которой он был определен. Существует несколько мощных методик программирования, которые задействуют такой вид замыканий вложенных функций, и их использование стало довольно распространенным явлением в программировании на JavaScript. Поначалу замыкания могут сбивать с толку, но для уверенной работы с ними важно хорошо их понимать.

Первый шаг к пониманию замыканий — анализ правил лексической области видимости для вложенных функций. Взгляните на следующий код:

```
let scope = "глобальная область видимости";   // Глобальная переменная
function checkscope() {
    let scope = "локальная область видимости"; // Локальная переменная
    function f() { return scope; } // Возвратить значение scope
    return f();
}
checkscope()                                   // => "локальная область видимости"
```

Функция `checkscope()` объявляет локальную переменную, после чего определяет и вызывает функцию, которая возвращает значение этой переменной. Вам должно быть совершенно ясно, почему вызов `checkscope()` возвращает строку `"локальная область видимости"`. А теперь давайте слегка изменим код. Можете ли вы сказать, что будет возвращать модифицированный код?

```
let scope = "глобальная область видимости";   // Глобальная переменная
function checkscope() {
   let scope = "локальная область видимости"; // Локальная переменная
   function f() { return scope; }             // Возвратить значение scope
   return f;
}
let s = checkscope()();                       // Что это возвратит?
```

В приведенном коде пара круглых скобок была перемещена из функции `checkscope()` за ее пределы. Вместо вызова вложенной функции и возвращения ее результата `checkscope()` возвращает сам объект вложенной функции. Что происходит, когда мы вызываем эту вложенную функцию (с помощью второй пары круглых скобок в последней строке кода) снаружи функции, в которой она была определена?

Вспомните фундаментальное правило лексической области видимости: функции JavaScript выполняются с применением области видимости, в которой они были определены. Вложенная функция `f()` определялась в области видимости, где переменная `scope` была привязана к значению `"локальная область видимости"`. Такая привязка по-прежнему действует при выполнении `f` независимо от того, откуда выполнение инициировано. Следовательно, последняя строка предыдущего кода возвращает `"локальная область видимости"`, а не `"глобальная область видимости"`. Говоря кратко, это удивительная и мощная природа замыканий: они захватывают привязки локальных переменных (и параметров) внешней функции, внутри которой они определены.

В подразделе 8.4.1 мы определяли функцию `uniqueInteger()`, которая использовала свойство самой функции для отслеживания следующего значения, подлежащего возврату. Недостаток такого подхода заключается в том, что изобилующий ошибками или вредоносный код мог бы сбрасывать счетчик либо устанавливать его в нецелочисленное значение, заставляя функцию `uniqueInteger()` нарушать "уникальную" или "целочисленную" часть своего контракта. Замыкания захватывают локальные переменные одиночного вызова функции и могут задействовать захваченные переменные как закрытое состояние. Ниже показано, как можно было бы переписать `uniqueInteger()` с применением "немедленно вызываемого выражения функции" для определения пространства имен и замыкания, которое использует это пространство имен, чтобы поддерживать состояние закрытым:

```
let uniqueInteger = (function() {  // Определить и вызвать
   let counter = 0;                // Закрытое состояние функции ниже
   return function() { return counter++; };
}());
uniqueInteger()  // => 0
uniqueInteger()  // => 1
```

Для понимания приведенного кода вы должны его внимательно прочитать. Поначалу первая строка кода выглядит как присваивание функции переменной `uniqueInteger`. На самом деле код определяет и вызывает (что подсказывает открывающая круглая скобка в первой строке) функцию, поэтому переменной

uniqueInteger присваивается возвращаемое значение функции. Если исследовать тело функции, то несложно заметить, что ее возвращаемым значением является еще одна функция. Именно этот объект вложенной функции присваивается uniqueInteger. Вложенная функция имеет доступ к переменным в своей области видимости и может работать с переменной counter, определенной во внешней функции. После возврата из внешней функции никакой другой код не может видеть переменную counter: внутренняя функция имеет к ней эксклюзивный доступ.

Закрытые переменные вроде counter не обязаны быть эксклюзивными в единственном замыкании: внутри одной внешней функции вполне возможно определить две и более вложенные функции и разделять ту же самую область видимости. Рассмотрим следующий код:

```
function counter() {
    let n = 0;
    return {
        count: function() { return n++; },
        reset: function() { n = 0; }
    };
}
let c = counter(), d = counter();    // Создать два счетчика
c.count()                 // => 0
d.count()                 // => 0: счетчики независимы
c.reset();                // Методы reset() и count() разделяют состояние
c.count()                 // => 0: потому что мы сбросили c
d.count()                 // => 1: d не сбрасывается
```

Функция counter() возвращает объект "счетчика", который имеет два метода: count() возвращает следующее целое число, а reset() сбрасывает внутреннее состояние. Первый момент, который нужно понять, связан с тем, что два метода разделяют доступ к закрытой переменной n. Второй момент, подлежащий осмыслению, заключается в том, что каждый вызов counter() создает новую область видимости, которая не зависит от областей видимости, применяемых предыдущими вызовами, и новую закрытую переменную внутри этой области видимости. Таким образом, если вы вызовите counter() дважды, то получите два объекта счетчика с разными закрытыми переменными. Вызов count() или reset() на одном объекте счетчика не влияет на другой объект счетчика.

Стоит отметить, что вы можете комбинировать методику замыканий с методами получения и установки. Следующая версия функции counter() представляет собой вариацию кода, который приводился в подразделе 6.10.6, но для закрытого состояния она использует замыкания вместо того, чтобы полагаться на обыкновенное свойство объекта:

```
function counter(n) {//Аргумент функции n является закрытой переменной
    return {
        // Метод получения свойства возвращает
        // и инкрементирует закрытую переменную счетчика
        get count() { return n++; },
        // Метод установки свойства не разрешает уменьшать значение n
```

```
        set count(m) {
            if (m > n) n = m;
            else throw Error("счетчик можно устанавливать только в
    большее значение");
        }
    };
}
let c = counter(1000);
c.count             // => 1000
c.count             // => 1001
c.count = 2000;
c.count             // => 2000
c.count = 2000;     // !Error: счетчик можно устанавливать только
                    // в большее значение
```

Обратите внимание, что в новой версии функции `counter()` для хранения закрытого состояния, разделяемого методами доступа к свойству, не объявляется локальная переменная, а просто применяется ее параметр `n`. Это позволяет коду, вызывающему `counter()`, указывать начальное значение закрытой переменной.

Пример 8.2 является обобщением разделяемого закрытого состояния посредством демонстрируемой здесь методики замыканий. В примере определяется функция `addPrivateProperty()`, которая определяет закрытую переменную и две вложенные функции для получения и установки данной переменной. Вложенные функции добавляются в виде методов к указываемому вами объекту.

Пример 8.2. Методы доступа к закрытому свойству, использующие замыкания

```
// Эта функция добавляет методы доступа для свойства с указанным
// именем объекта o.
// Методы именуются как get<name> и set<name>.
// Если предоставляется функция предиката, тогда метод установки применяет
// ее для проверки своего аргумента на предмет допустимости перед его
// сохранением. Если предикат возвращает false, то метод установки
// генерирует исключение.
//
// Необычной особенностью этой функции является то, что значение свойства,
// которым манипулируют методы получения и установки, не сохраняется в объекте o.
// Взамен значение хранится только в локальной переменной в данной функции.
// Методы получения и установки также определяются локально внутри функции
// и потому имеют доступ к этой локальной переменной.
// Таким образом, значение закрыто по отношению к двум методам доступа и его
// невозможно устанавливать или модифицировать иначе, чем через метод установки.
function addPrivateProperty(o, name, predicate) {
    let value;    // Значение свойства

    // Метод получения просто возвращает value
    o[`get${name}`] = function() { return value; };

    // Метод установки сохраняет значение или генерирует
    // исключение, если предикат отклоняет значение
    o[`set${name}`] = function(v) {
```

```
        if (predicate && !predicate(v)) {
            throw new TypeError(`set${name}: недопустимое значение ${v}`);
        } else {
            value = v;
        }
    };
}
// В следующем коде демонстрируется работа метода addPrivateProperty()
let o = {};         // Пустой объект

// Добавить методы доступа к свойству getName() и setName().
// Удостовериться, что разрешены только строковые значения.
addPrivateProperty(o, "Name", x => typeof x === "string");

o.setName("Frank"); // Установить значение свойства
o.getName()         // => "Frank"
o.setName(0);       // !TypeError: попытка установки значения неправильного типа
```

К настоящему времени было показано несколько примеров, в которых два замыкания определялись в одной и той же области видимости и разделяли доступ к той же самой закрытой переменной или переменным. Описанная методика важна, но не менее важно распознавать ситуации, когда замыкания непреднамеренно разделяют доступ к переменной, чего быть не должно. Взгляните на приведенный далее код:

```
// Эта функция возвращает функцию, которая всегда возвращает v
function constfunc(v) { return () => v; }

// Создать массив константных функций:
let funcs = [];
for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);

// Функция в элементе 5 массива возвращает значение 5:
funcs[5]()    // => 5
```

При работе с кодом подобного рода, который создает множество замыканий в цикле, распространенной ошибкой будет попытка переноса цикла внутрь функции, определяющей замыкания. Скажем, подумайте о таком коде:

```
// Возвращает массив функций, которые возвращают значения 0-9
function constfuncs() {
    let funcs = [];
    for(var i = 0; i < 10; i++) {
        funcs[i] = () => i;
    }
    return funcs;
}

let funcs = constfuncs();
funcs[5]()  // => 10; почему не возвратилось значение 5?
```

Код создает 10 замыканий и сохраняет их в массиве. Все замыкания определены внутри того же самого вызова функции, поэтому они разделяют доступ к переменной `i`. Когда происходит возврат из функции `constfuncs()`, значение

переменной i равно 10, и его разделяют все 10 замыканий. Следовательно, все функции в возвращенном массиве функций возвращают то же самое значение, приводя совершенно не к тому результату, который был желателен. Важно помнить о том, что область видимости, ассоциированная с замыканием, является "динамической". Вложенные функции не создают закрытые копии области видимости или статические снимки привязок переменных. По существу проблема здесь в том, что переменные, объявленные с помощью var, определены повсюду в функции. Наш цикл for объявляет переменную цикла посредством var i, поэтому переменная i определена везде в функции, а не имеет суженную область видимости внутри тела цикла. В коде демонстрируется распространенная категория ошибок в ES5 и предшествующих версиях, но введение переменных с блочной областью видимости в ES6 решает проблему. Если мы просто поменяем var на let или const, то проблема исчезнет. Поскольку let и const дают блочную область видимости, каждая итерация цикла определяет область видимости, независимую от областей видимости для всех остальных итераций, и каждая область видимости имеет собственную независимую привязку i.

Еще один момент, о котором нужно помнить при написании замыканий, касается того, что this — это ключевое слово JavaScript, а не переменная. Как обсуждалось ранее, стрелочные функции наследуют значение this функции, которая их содержит, но функции, определенные с помощью ключевого слова function, нет. Таким образом, если вы пишете замыкание, в котором необходимо применять значение this содержащей функции, то должны использовать стрелочную функцию, вызвать bind() на замыкании перед его возвращением или присвоить внешнее значение this переменной, которую унаследует ваше замыкание:

```
const self = this; //Сделать значение this доступным вложенным функциям
```

8.7. Свойства, методы и конструктор функций

Мы видели, что функции являются значениями в программах JavaScript. Операция typeof в случае применения к функции возвращает строку "function", но в действительности функции представляют собой специализированный вид объекта JavaScript. Поскольку функции — это объекты, как любой другой объект они могут иметь свойства и методы. Существует даже конструктор Function() для создания новых объектов функций. В последующих подразделах будут описаны свойства length, name и prototype, методы call(), apply(), bind() и toString(), а также конструктор Function().

8.7.1. Свойство length

Предназначенное только для чтения свойство length функции указывает ее *арность* — количество параметров, объявленное в списке параметров функции, которое обычно соответствует количеству аргументов, ожидаемых функцией. Если функция имеет параметр остатка, то в свойстве length он не учитывается.

8.7.2. Свойство name

Предназначенное только для чтения свойство name функции указывает имя, которое использовалось при определении функции, если она была определена с именем, либо имя переменной или свойства, которому было присвоено выражение неименованной функции, когда оно создавалось в первый раз. Свойство name главным образом полезно при написании отладочных сообщений и сообщений об ошибках.

8.7.3. Свойство prototype

Все функции кроме стрелочных имеют свойство prototype, которое ссылается на объект, известный как *объект прототипа*. Каждая функция имеет отличающийся объект прототипа. Когда функция применяется в качестве конструктора, вновь созданный объект наследует свойства от объекта прототипа. Прототипы и свойство prototype обсуждались в подразделе 6.2.3, и будут подробно рассматриваться в главе 9.

8.7.4. Методы call() и apply()

Методы call() и apply() позволяют косвенно вызывать (см. подраздел 8.2.4) функцию, как если бы она была методом какого-то другого объекта. Первым аргументом в call() и apply() является объект, на котором должна вызываться функция; он представляет контекст вызова и становится значением ключевого слова this внутри тела функции. Чтобы вызвать функцию f() как метод объекта o (без аргументов), вы можете использовать либо call(), либо apply():

```
f.call(o);
f.apply(o);
```

Любая из двух строк подобна следующему коду (где предполагается, что объект o не имеет свойства по имени m):

```
o.m = f;          // Сделать f временным методом o
o.m();            // Вызвать его без аргументов
delete o.m;       // Удалить временный метод
```

Вспомните, что стрелочные функции наследуют значение this контекста, где они определены. Это не удастся переопределить посредством методов call() и apply(). В случае вызова их на стрелочной функции первый аргумент фактически игнорируется.

Любые аргументы call() после первого аргумента с контекстом вызова будут значениями, которые передаются вызываемой функции (и такие аргументы не игнорируются для стрелочных функций). Например, чтобы передать два числа функции f() и вызвать ее, как если бы она была методом объекта o, можно применять такой код:

```
f.call(o, 1, 2);
```

Метод apply() похож на метод call(), но аргументы, подлежащие передаче в функцию, указываются в виде массива:

```
f.apply(o, [1,2]);
```

Если функция определена для приема произвольного количества аргументов, то метод apply() позволяет вызывать ее с содержимым массива произвольной длины. В ES6 и последующих версиях мы можем просто использовать операцию распространения, но вы можете встретить код ES5, в котором взамен применяется apply(). Скажем, чтобы найти наибольшее число в массиве чисел, не используя операцию распространения, мы могли бы применить метод apply() для передачи элементов массива функции Math.max():

```
let biggest = Math.max.apply(Math, arrayOfNumbers);
```

Функция trace(), которая определена ниже, подобна функции timed(), определенной в подразделе 8.3.4, но работает для методов, а не для функций. Вместо операции распространения в ней применяется метод apply() и за счет этого появляется возможность вызова заключенного в оболочку метода с теми же аргументами и значением this, что и у метода-оболочки:

```
// Заменить метод по имени m объекта o версией, которая записывает
// в журнал сообщения до и после вызова исходного метода.
function trace(o, m) {
   let original = o[m];          // Запомнить исходный метод в замыкании
   o[m] = function(...args) {    // Определить новый метод
      console.log(new Date(), "Entering:", m);  // Записать сообщение
                                                // в журнал
      let result = original.apply(this, args);  // Вызвать исходный
                                                // метод
      console.log(new Date(), "Exiting:", m);   // Записать сообщение
                                                // в журнал
      return result;            // Возвратить результат
   };
}
```

8.7.5. Метод bind()

Основная цель метода bind() — *привязка* функции к объекту. В результате вызова метода bind() на функции f и передача объекта o возвращается новая функция. Вызов новой функции (как функции) приводит к вызову исходной функции f как метода объекта o. Любые аргументы, передаваемые новой функции, передаются исходной функции. Вот пример:

```
function f(y) { return this.x + y; } //Этой функции необходима привязка
let o = { x: 1 };  // Объект, к которому будет осуществляться привязка
let g = f.bind(o);      // Вызов g(x) приводит к вызову f() на o
g(2)                    // => 3
let p = { x: 10, g };   // Вызов g() как метода объекта this
p.g(2)                  // => 3: g по-прежнему привязан к o, не к p
```

Стрелочные функции наследуют значение this от среды, в которой они определяются, и это значение не может быть переопределено с помощью bind(), так что если функция f() в предыдущем коде была определена как стрелочная, то привязка работать не будет. Однако наиболее часто bind() используется для того, чтобы заставить функции, не являющиеся стрелочными, вести себя подобно стрелочным, а потому такое ограничение на практике не создает проблем.

Тем не менее, метод bind() не только привязывает функцию к объекту. Он также обеспечивает частичное применение: любые аргументы, переданные bind() после первого, привязываются наряду со значением this. Средство частичного применения, поддерживаемое методом bind(), работает со стрелочными функциями. Частичное применение — распространенная методика в функциональном программировании, которую иногда называют *каррированием* или *каррингом* (currying). Ниже показаны примеры использования bind() для частичного применения:

```
let sum = (x,y) => x + y;          // Возвращает сумму двух аргументов
let succ = sum.bind(null, 1);      // Привязывает первый аргумент к 1
succ(2) // => 3: x привязывается к 1, а для аргумента y передается 2

function f(y,z) { return this.x + y + z; }
let g = f.bind({x: 1}, 2);         // Привязывает this и y
g(3)   // => 6: this.x привязывается к 1, y привязывается к 2 и z - к 3
```

Свойство name функции, возвращенной методом bind(), будет свойством name функции, на которой вызывался bind(), снабженным префиксом "bound".

8.7.6. Метод toString()

Как и все объекты JavaScript, функции имеют метод toString(). Спецификация ECMAScript требует, чтобы метод toString() возвращал строку, которая следует синтаксису оператора объявления функции. На практике большинство (но не все) реализаций метода toString() возвращают полный исходный код функции. Метод toString() встроенных функций обычно возвращает строку, которая включает что-то вроде "[native code]" в качестве тела функции.

8.7.7. Конструктор Function()

Поскольку функции являются объектами, существует конструктор Function(), который можно использовать для создания новых функций:

```
const f = new Function("x", "y", "return x*y;");
```

Такая строка кода создает новую функцию, которая более или менее эквивалентна функции, определенной с помощью знакомого синтаксиса:

```
const f = function(x, y) { return x*y; };
```

Конструктор Function() ожидает передачи любого количества строковых аргументов. Последним аргументом должен быть текст тела функции; он может содержать произвольные операторы JavaScript, отделенные друг от друга точ-

ками с запятой. Все остальные аргументы, передаваемые конструктору, представляют собой строки, которые указывают имена параметров для функции. В случае определения функции без аргументов понадобится просто передать конструктору единственную строку — тело функции.

Обратите внимание, что конструктору `Function()` не передается аргумент, в котором указывалось бы имя создаваемой функции. Подобно литералам типа функций конструктор `Function()` создает анонимные функции.

Ниже перечислено несколько моментов, касающихся конструктора `Function()`, которые важно понимать.

- Конструктор `Function()` позволяет динамически создавать и компилировать во время выполнения функции JavaScript.
- Конструктор `Function()` производит разбор тела функции и создает новый объект функции каждый раз, когда он вызывается. Если вызов конструктора располагается в цикле или внутри часто вызываемой функции, тогда такой процесс может быть неэффективным. Напротив, вложенные функции и выражения функций, находящиеся внутри циклов, не компилируются заново каждый раз, когда они встречаются.
- Последний очень важный момент, связанный с конструктором `Function()`, заключается в том, что создаваемые им функции не используют лексическую область видимости; взамен они всегда компилируются, как если бы они были функциями верхнего уровня, что и демонстрируется в следующем коде:

```
let scope = "глобальная область видимости";
function constructFunction() {
    let scope = "локальная область видимости";
    return new Function("return scope"); // Не захватывает локальную
                                         // переменную scope!
}
// Этот код возвращает строку "глобальная область видимости",
// потому что функция,
// возвращаемая конструктором Function(),
// не использует локальную переменную scope.
constructFunction()()  // => "глобальная"
```

Конструктор `Function()` лучше всего воспринимать как глобально ограниченную версию функции `eval()` (см. подраздел 4.12.2), которая определяет переменные и функции в собственной закрытой области видимости. Возможно, вам никогда не придется применять конструктор `Function()` в своем коде.

8.8. Функциональное программирование

JavaScript не является языком функционального программирования, подобным Lisp или Haskell, но тот факт, что JavaScript позволяет манипулировать функциями как объектами, означает возможность использования мето-

дик функционального программирования. Методы массивов вроде map() и reduce() особенно хорошо подходят к стилю функционального программирования. В последующих подразделах обсуждаются методики функционального программирования в JavaScript. Они задуманы как расширенное исследование мощи функций JavaScript, а не в качестве рекомендации хорошего стиля программирования.

8.8.1. Обработка массивов с помощью функций

Пусть у нас есть массив чисел, и мы хотим рассчитать для них среднюю величину и стандартное отклонение. Мы могли бы решить задачу в нефункциональном стиле:

```
let data = [1,1,3,5,5];         // Массив чисел

// Среднее mean представляет собой сумму элементов,
// деленную на их количество
let total = 0;
for(let i = 0; i < data.length; i++) total += data[i];
let mean = total/data.length; // mean == 3; среднее наших данных равно 3

// Для расчета стандартного отклонения мы суммируем
// квадраты отклонений элементов от среднего
total = 0;
for(let i = 0; i < data.length; i++) {
   let deviation = data[i] - mean;
   total += deviation * deviation;
}
let stddev = Math.sqrt(total/(data.length-1));  // stddev == 2
```

Те же самые вычисления мы можем выполнить в лаконичном функциональном стиле с применением методов массивов map() и reduce() (которые рассматривались в подразделе 7.8.1):

```
// Первым делом определить две простых функции
const sum = (x,y) => x+y;
const square = x => x*x;

// Затем использовать эти функции с методами класса Array
// для вычисления средней величины и стандартного отклонения
let data = [1,1,3,5,5];
let mean = data.reduce(sum)/data.length;          // mean == 3
let deviations = data.map(x => x-mean);
let stddev =
    Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1));
```

Новая версия кода выглядит совсем не так, как первая, но в ней по-прежнему вызываются методы на объектах, оставляя ряд соглашений из объектно-ориентированного программирования. Давайте напишем функциональные версии методов map() и reduce():

```
const map = function(a, ...args) { return a.map(...args); };
const reduce = function(a, ...args) { return a.reduce(...args); };
```

При наличии определенных выше функций map() и reduce() наш код для расчета средней величины и стандартного отклонения приобретает следующий вид:

```
const sum = (x,y) => x+y;
const square = x => x*x;

let data = [1,1,3,5,5];
let mean = reduce(data, sum)/data.length;
let deviations = map(data, x => x-mean);
let stddev=Math.sqrt(reduce(map(deviations,square),sum)/(data.length-1));
stddev  // => 2
```

8.8.2. Функции высшего порядка

Функция высшего порядка — это функция, которая оперирует функциями, принимая одну или большее количество функций в качестве аргументов и возвращая новую функцию. Вот пример:

```
// Эта функция высшего порядка возвращает новую функцию,
// которая передает свои аргументы f и возвращает логическое
// отрицание возвращаемого значения f
function not(f) {
   return function(...args) {           // Возвратить новую функцию,
      let result = f.apply(this, args); // которая вызывает f
      return !result;                   // и выполняет логическое
                                        // отрицание ее результата.
   };
}
const even = x => x % 2 === 0;   // Функция для определения,
                                 // четное ли число
const odd = not(even);           // Новая функция, которая делает
                                 // противоположное
[1,1,3,5,5].every(odd)           // => true: каждый элемент массива
                                 // является нечетным
```

Функция not() является функцией высшего порядка, т.к. она принимает аргумент типа функции и возвращает новую функцию. В качестве другого примера рассмотрим приведенную ниже функцию mapper(). Она принимает аргумент типа функции и возвращает новую функцию, которая отображает один массив на другой, применяя переданную функцию. Функция mapper() использует определенную ранее функцию, и важно понимать, чем отличаются эти две функции:

```
// Возвращает функцию, которая ожидает аргумент типа массива и приме-
// няет f к каждому элементу, возвращая массив возвращаемых значений.
// Сравните ее с определенной ранее функцией map().
function mapper(f) {
   return a => map(a, f);
}
const increment = x => x+1;
const incrementAll = mapper(increment);
incrementAll([1,2,3])  // => [2,3,4]
```

Вот еще один, более универсальный пример функции высшего порядка, которая принимает две функции, f и g, а возвращает новую функцию, вычисляющую f(g()):

```
// Возвращает новую функцию, которая вычисляет f(g(...)).
// Возвращаемая функция h передает все свои аргументы функции g,
// далее передает возвращаемое значение g функции f и затем
// возвращает возвращаемое значение f.
// Функции f и g вызываются с тем же значением this,
// с каким вызывалась h.
function compose(f, g) {
   return function(...args) {
    // Мы используем call для f, потому что передаем одиночное значение,
    // и apply для g, поскольку передаем массив значений.
    return f.call(this, g.apply(this, args));
   };
}
const sum = (x,y) => x+y;
const square = x => x*x;
compose(square, sum)(2,3)  // => 25; квадрат суммы
```

Функции partial() и memoize(), определяемые в подразделах ниже, являются двумя более важными функциями высшего порядка.

8.8.3. Функции с частичным применением

Метод bind() функции f (см. подраздел 8.7.5) возвращает новую функцию, которая вызывает f в указанном контексте и с заданным набором аргументов. Мы говорим, что он привязывает функцию к объекту и частично применяет аргументы. Метод bind() частично применяет аргументы слева — т.е. аргументы, передаваемые методу bind(), помещаются в начало списка аргументов, который передается исходной функции. Но также имеется возможность частично применять аргументы справа:

```
// Аргументы этой функции передаются слева
function partialLeft(f, ...outerArgs) {
   return function(...innerArgs) {  // Возвратить эту функцию
      let args = [...outerArgs, ...innerArgs];   // Построить список
                                                 // аргументов
      return f.apply(this, args);   // Затем вызвать с ним функцию f
   };
}

// Аргументы этой функции передаются справа
function partialRight(f, ...outerArgs) {
   return function(...innerArgs) {  // Возвратить эту функцию
      let args = [...innerArgs, ...outerArgs];   // Построить список
                                                 // аргументов
      return f.apply(this, args);   // Затем вызвать с ним функцию f
   };
}
```

```
// Аргументы этой функции служат шаблоном. Неопределенные значения
// в списке аргументов заполняются значениями из внутреннего набора.
function partial(f, ...outerArgs) {
   return function(...innerArgs) {
      let args = [...outerArgs];  // Локальная копия шаблона внешних
                                  // аргументов
      let innerIndex=0;  // Какой внутренний аргумент будет следующим
      // Проход в цикле по аргументам с заполнением неопределенных
      // значений значениями из внутренних аргументов
      for(let i = 0; i < args.length; i++) {
         if (args[i] === undefined) args[i] = innerArgs[innerIndex++];
      }
      // Присоединить любые оставшиеся внутренние аргументы
      args.push(...innerArgs.slice(innerIndex));
      return f.apply(this, args);
   };
}
// Функция с тремя аргументами
const f = function(x,y,z) { return x * (y - z); };
// Обратите внимание на отличия этих трех частичных применений
partialLeft(f, 2)(3,4) // => -2: Привязывает первый аргумент: 2 * (3 - 4)
partialRight(f, 2)(3,4)       // => 6: Привязывает последний аргумент:
                              // 3 * (4 - 2)
partial(f, undefined, 2)(3,4) // => -6: Привязывает средний аргумент:
                              // 3 * (2 - 4)
```

Такие функции с частичным применением позволяют нам легко определять интересные функции из функций, которые мы уже определили. Ниже приведено несколько примеров:

```
const increment = partialLeft(sum, 1);
const cuberoot = partialRight(Math.pow, 1/3);
cuberoot(increment(26))       // => 3
```

Частичные применения становятся даже более интересными, когда мы комбинируем их с другими функциями высшего порядка. Например, далее демонстрируется способ определения предыдущей функции `not()` с использованием композиции и частичного применения:

```
const not = partialLeft(compose, x => !x);
const even = x => x % 2 === 0;
const odd = not(even);
const isNumber = not(isNaN);
odd(3) && isNumber(2)         // => true
```

Композицию и частичное применение мы можем использовать также для переделки расчетов средней величины и стандартного отклонения в исключительно функциональном стиле:

```
// Функции sum() и square() были определены ранее. Вот еще несколько:
const product = (x,y) => x*y;
const neg = partial(product, -1);
const sqrt = partial(Math.pow, undefined, .5);
const reciprocal = partial(Math.pow, undefined, neg(1));
```

```
// А теперь вычислим среднюю величину и стандартное отклонение:
let data = [1,1,3,5,5];  // Наши данные
let mean = product(reduce(data, sum), reciprocal(data.length));
let stddev = sqrt(product(reduce(map(data,
                                     compose(square,
                                             partial(sum, neg(mean)))), sum),
                          reciprocal(sum(data.length,neg(1)))));
[mean, stddev]    // => [3, 2]
```

Обратите внимание, что код для вычисления средней величины и стандартного отклонения представляет собой целиком вызовы функций; операции не задействованы, а количество круглых скобок стало настолько большим, что приведенный выше код JavaScript начинает быть похожим на код Lisp. Повторюсь, это не тот стиль, который я рекомендую для программирования на JavaScript, но упражнение интересно тем, что позволяет увидеть, до какой степени глубоко функциональным может быть код JavaScript.

8.8.4. Мемоизация

В подразделе 8.4.1 мы определили функцию вычисления факториала, которая кешировала свои ранее вычисленные результаты. В функциональном программировании такой вид кеширования называется *мемоизацией* (memoization). В следующем коде показана функция высшего порядка `memoize()`, которая принимает функцию в качестве аргумента и возвращает мемоизованную версию функции:

```
// Возвращает мемоизованную версию f. Работает, только если все
// аргументы f имеют отличающиеся строковые представления.
function memoize(f) {
   const cache = new Map();  // Кеш значений хранится в замыкании

   return function(...args) {
      // Создать строковую версию аргументов для использования
      // в качестве ключа кеша
      let key = args.length + args.join("+");
      if (cache.has(key)) {
         return cache.get(key);
      } else {
         let result = f.apply(this, args);
         cache.set(key, result);
         return result;
      }
   };
}
```

Функция `memoize()` создает новый объект для кеша и присваивает его локальной переменной, чтобы он был закрытым внутри (в замыкании) возвращаемой функции. Возвращаемая функция преобразует массив своих аргументов в строку и применяет эту строку как имя свойства для объекта кеша. Если значение присутствует в кеше, тогда оно возвращается напрямую. В противном случае она вызывает указанную функцию, чтобы вычислить значение для этих аргументов, кеширует результирующее значение и возвращает его. Вот как можно было бы использовать `memoize()`:

```
// Возвращает наибольший общий делитель двух целых чисел, используя
// алгоритм Евклида: https://ru.wikipedia.org/wiki/Алгоритм_Евклида
function gcd(a,b) {     // Проверка типов для a и b не показана
   if (a < b) {         // В начале удостовериться, что a >= b
      [a, b] = [b, a]; // Деструктурирующее присваивание, чтобы
                       // поменять местами переменные
   }
   while(b !== 0) {     // Алгоритм Евклида для нахождения наибольшего
                       // общего делителя
      [a, b] = [b, a%b];
   }
   return a;
}
const gcdmemo = memoize(gcd);
gcdmemo(85, 187)  // => 17
// Обратите внимание, что при написании рекурсивной функции,
// которая будет мемоизоваться,
// мы обычно хотим возвращаться к мемоизованной версии,
// а не к исходной.
const factorial = memoize(function(n) {
   return (n <= 1) ? 1 : n * factorial(n-1);
});
factorial(5)      // => 120: также кеширует значения для 4, 3, 2 и 1.
```

8.9. Резюме

Ниже перечислены основные моменты, рассмотренные в главе, которые следует запомнить.

- Вы можете определять функции с помощью ключевого слова `function` и синтаксиса стрелочных функций `=>` в ES6.
- Вы можете вызывать функции, которые могут применяться как методы и конструкторы.
- Некоторые средства ES6 позволяют определять стандартные значения для необязательных параметров функций, собирать множество аргументов в массив, используя параметр остатка, а также деструктурировать аргументы типа объектов и массивов в параметры функций.
- Вы можете применять операцию распространения `...` для передачи элементов массива или другого итерируемого объекта в качестве аргументов в вызове функции.
- Функция, определенная внутри вмещающей функции и возвращаемая ею, предохраняет доступ к своей лексической области видимости и потому может читать и записывать переменные, которые определены во внешней функции. Функции, используемые подобным образом, называются замыканиями, и полезно понимать методику работы с ними.
- Функции являются объектами, которыми можно манипулировать в коде JavaScript, что делает возможным функциональный стиль программирования.

ГЛАВА 9

Классы

Объекты JavaScript раскрывались в главе 6, где каждый объект трактовался как уникальный набор свойств, отличающийся от любого другого объекта. Однако часто бывает полезно определять *класс* объектов, которые разделяют определенные свойства. Члены, или *экземпляры*, класса имеют собственные свойства для хранения или определения их состояния, но они также располагают методами, которые устанавливают их поведение. Такие методы определяются классом и разделяются всеми экземплярами. Например, вообразим себе класс по имени `Complex`, который представляет и выполняет арифметические действия с комплексными числами. Экземпляр `Complex` мог бы иметь свойства для хранения действительной и мнимой частей (состояния) комплексного числа. Вдобавок в классе `Complex` были бы определены методы для выполнения сложения и умножения (поведение) комплексных чисел.

Классы в JavaScript используют наследование, основанное на прототипах: если два объекта наследуют свойства (как правило, свойства-функции, или методы) из одного и того же прототипа, то мы говорим, что эти объекты являются экземплярами того же самого класса. Так в двух словах работают классы JavaScript. Прототипы и наследование JavaScript обсуждались в подразделах 6.2.3 и 6.3.2, и вы должны быть хорошо знакомы с тем, что там излагалось, чтобы понять материал настоящей главы. Прототипы рассматриваются в разделе 9.1.

Если два объекта наследуются от одного и того же прототипа, то это обычно (но не обязательно) означает, что они были созданы и инициализированы той же самой функцией конструктора или фабричной функцией. Конструкторы раскрывались в разделе 4.6 и подразделах 6.2.2 и 8.2.3 и продолжат обсуждаться в разделе 9.2.

В JavaScript всегда было разрешено определять классы. В ES6 появился совершенно новый синтаксис (в том числе ключевое слово `class`), который еще больше облегчает создание классов. Новые классы JavaScript действуют аналогично классам старого стиля, и глава начинается с объяснения старого способа создания классов, поскольку он более четко показывает, что происходит "за кулисами" для того, чтобы обеспечить работу классов. После изложения основ мы приступим к применению нового упрощенного синтаксиса определения классов.

Если вы знакомы со строго типизированными языками объектно-ориентированного программирования вроде Java или C++, тогда заметите, что классы JavaScript сильно отличаются от классов в упомянутых языках. Имеется ряд синтаксических сходств и в JavaScript есть возможность эмулировать многие средства "классических" классов, но лучше сразу осознать, что классы и механизм наследования на основе прототипов JavaScript значительно отличается от классов и механизма наследования, основанного на классах, в Java и похожих языках.

9.1. Классы и прототипы

Класс в JavaScript представляет собой набор объектов, которые наследуют свойства от того же самого объекта прототипа. Следовательно, объект прототипа является главной характерной чертой класса. В главе 6 рассматривалась функция `Object.create()`, которая возвращает вновь созданный объект, унаследованный от указанного объекта прототипа. Если мы определим объект прототипа и затем используем функцию `Object.create()` для создания унаследованных от него объектов, то определим класс JavaScript. В большинстве случаев экземпляры класса требуют дальнейшей инициализации, поэтому обычно определяют функцию, которая создает и инициализирует новый объект. Прием демонстрируется в примере 9.1: в нем определяется объект прототипа для класса, представляющего диапазон значений, и также определяется *фабричная функция*, которая создает и инициализирует новый экземпляр данного класса.

Пример 9.1. Простой класс JavaScript

```
// Фабричная функция, которая возвращает новый объект диапазона.
function range(from, to) {
    // Использовать Object.create() для создания объекта, который наследуется
    // от объекта прототипа, определенного ниже. Объект прототипа хранится как
    // свойство этой функции и определяет разделяемые методы (поведение)
    // для всех объектов, представляющих диапазоны.
    let r = Object.create(range.methods);

    // Сохранить начальную и конечную точки (состояние) нового объекта диапазона.
    // Эти свойства не являются унаследованными и они уникальны для этого объекта.
    r.from = from;
    r.to = to;

    // В заключение возвратить новый объект.
    return r;
}

// Объект прототипа, определяющий свойства, которые наследуются
// всеми объектами, представляющими диапазоны.
range.methods = {
    // Возвращает true, если x входит в диапазон, и false в противном случае.
    // Метод работает как с числовыми, так и с текстовыми диапазонами
    // и диапазонами Date.
    includes(x) { return this.from <= x && x <= this.to; },
```

```
  // Генераторная функция, которая делает экземпляры класса итерируемыми.
  // Обратите внимание, что она работает только с числовыми диапазонами.
  *[Symbol.iterator]() {
    for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
  },
  // Возвращает строковое представление диапазона.
  toString() { return "(" + this.from + "..." + this.to + ")"; }
};

// Пример использования объекта диапазона.
let r = range(1,3);     // Создать объект диапазона
r.includes(2)           // => true: 2 входит в диапазон
r.toString()            // => "(1...3)"
[...r]                  // => [1, 2, 3]; преобразовать в массив через итератор
```

В коде примера 9.1 полезно отметить несколько моментов.

- В коде определяется фабричная функция `range()` для создания новых объектов, представляющих диапазоны.
- В коде применяется свойство `methods` функции `range()` как удобное место для сохранения объекта прототипа, который определяет класс. В размещении здесь объекта прототипа нет ничего особенного или идиоматического.
- Функция `range()` определяет свойства `from` и `to` в каждом объекте диапазона. Эти свойства не являются унаследованными, они не используются совместно и задают уникальное состояние каждого отдельного объекта диапазона.
- Для определения методов объект `range.methods` применяет сокращенный синтаксис ES6, из-за чего вы нигде не видите ключевое слово `function`. (Сокращенный синтаксис определения методов в объектных литералах рассматривался в подразделе 6.10.5.)
- Один из методов в прототипе имеет вычисляемое имя (см. подраздел 6.10.2) `Symbol.iterator` и потому определяет итератор для объектов, представляющих диапазоны. Имя метода предварено символом *, который служит признаком генераторной, а не обыкновенной функции. Итераторы и генераторы будут подробно раскрыты в главе 12. Пока достаточно знать, что в результате экземпляры класса диапазона могут использоваться с циклом `for/of` и с операцией распространения `...`.
- Все разделяемые унаследованные методы, определенные в `range.methods`, работают со свойствами `from` и `to`, которые были инициализированы фабричной функцией `range()`. Для ссылки на них применяется ключевое слово `this`, чтобы обращаться к объекту, через который они были вызваны. Такое использование `this` является фундаментальной характеристикой методов любого класса.

9.2. Классы и конструкторы

В примере 9.1 демонстрируется простой прием определения класса JavaScript. Тем не менее, это не идиоматический способ достижения цели, потому что не был определен *конструктор*. Конструктор представляет собой функцию, предназначенную для инициализации вновь созданных объектов. Как объяснялось в подразделе 8.2.3, конструкторы вызываются с применением ключевого слова `new`.

Вызовы конструкторов, использующие `new`, автоматически создают новый объект, так что самому конструктору необходимо лишь инициализировать состояние нового объекта. Важной особенностью вызовов конструкторов является то, что свойство `prototype` конструктора применяется в качестве прототипа нового объекта. В подразделе 6.2.3 были введены прототипы и сделан особый акцент на том, что хотя почти все объекты имеют прототип, только некоторые располагают свойством `prototype`.

Наконец-то мы можем прояснить сказанное: свойство `prototype` имеется у объектов функций. Это означает, что все объекты, созданные с помощью одной и той же функции конструктора, унаследованы от того же самого объекта и в итоге являются членами того же самого класса. В примере 9.2 показано, как можно было бы изменить класс диапазона из примера 9.1, чтобы вместо фабричной функции использовать функцию конструктора. В примере 9.2 демонстрируется идиоматический способ создания класса в версиях JavaScript, которые не поддерживают ключевое слово `class` из ES6. Несмотря на то что ключевое слово `class` теперь хорошо поддерживается, все еще существует много более старого кода JavaScript, где классы определяются подобным образом, и вы должны быть знакомы с такой идиомой, чтобы иметь возможность читать старый код и понимать происходящее "за кулисами", когда применяется ключевое слово `class`.

Пример 9.2. Класс Range, использующий конструктор

```
// Функция конструктора, которая инициализирует новые объекты Range.
// Обратите внимание, что она не создает и не возвращает объект,
// а просто инициализирует this.
function Range(from, to) {
    // Сохранить начальную и конечную точки (состояние) нового объекта диапазона.
    // Эти свойства не являются унаследованными и они уникальны для этого объекта.
    this.from = from;
    this.to = to;
}

// От этого объекта наследуются все объекты Range. Обратите внимание,
// что именем свойства должно быть prototype, чтобы это работало.
Range.prototype = {
    // Возвращает true, если x входит в диапазон, и false в противном случае.
    // Метод работает как с числовыми, так и с текстовыми диапазонами
    // и диапазонами Date.
    includes: function(x) { return this.from <= x && x <= this.to; },
```

```
  // Генераторная функция, которая делает экземпляры класса итерируемыми.
  // Обратите внимание, что она работает только с числовыми диапазонами.
  [Symbol.iterator]: function*() {
    for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
  },
  // Возвращает строковое представление диапазона.
  toString: function() { return "(" + this.from + "..." + this.to + ")"; }
};

// Пример использования нового класса Range.
let r = new Range(1,3); // Создать объект Range; здесь применяется new
r.includes(2)           // => true: 2 входит в диапазон
r.toString()            // => "(1...3)"
[...r]                  // => [1, 2, 3]; преобразовать в массив через итератор
```

Полезно тщательно сравнить примеры 9.1 и 9.2, отметив отличия между двумя методиками для определения классов. Прежде всего, обратите внимание, что мы переименовали фабричную функцию `range()` в `Range()`, когда преобразовывали ее в конструктор. Это очень распространенное соглашение при написании кода: функции конструкторов в известном смысле определяют классы, а классы имеют имена, которые (по соглашению) начинаются с заглавных букв. Имена обыкновенных функций и методов начинаются со строчных букв.

Далее обратите внимание, что конструктор `Range()` вызывается (в конце примера) с ключевым словом `new`, в то время как фабричная функция `range()` вызывалась без него. В примере 9.1 использовался вызов обыкновенной функции (см. подраздел 8.2.1), чтобы создать новый объект, а в примере 9.2 применялся вызов конструктора (см. подраздел 8.2.3). Поскольку конструктор `Range()` вызывается с `new`, он не обязан вызывать `Object.create()` или предпринимать любое другое действие для создания нового объекта. Новый объект создается автоматически перед вызовом конструктора и доступен как значение `this`.

Конструктор `Range()` просто должен инициализировать `this`. Конструкторы даже не обязаны возвращать вновь созданный объект. Вызов конструктора автоматически создает новый объект, вызывает конструктор как метод этого объекта и возвращает новый объект.

Тот факт, что вызов конструктора настолько отличается от вызова обыкновенной функции, является еще одной причиной назначения конструкторам имен, начинающихся с заглавных букв. Конструкторы написаны так, чтобы вызываться как конструкторы, с ключевым словом `new`, и они обычно не будут работать надлежащим образом, если вызываются как обыкновенные функции. Соглашение об именовании, которое отделяет функции конструкторов от обыкновенных функций, помогает программистам распознавать, когда должно использоваться `new`.

Конструкторы и new.target

Внутри тела функции вы можете выяснить, была ли функция вызвана как конструктор, с помощью специального выражения `new.target`. Если значение этого выражения определено, тогда вы знаете, что функция вызывалась как конструктор, с ключевым словом `new`. Во время обсуждения подклассов в разделе 9.5 вы увидите, что выражение `new.target` не всегда оказывается ссылкой на конструктор, в котором оно применяется: оно также может ссылаться на функцию конструктора подкласса.

Если выражение `new.target` равно `undefined`, тогда вмещающая функция вызывалась как функция, без ключевого слова `new`. Разнообразные конструкторы объектов ошибок JavaScript могут вызываться без `new`, и если вы хотите эмулировать такую особенность в своих конструкторах, то можете записывать их следующим образом:

```
function C() {
    if (!new.target) return new C();
    // код инициализации
}
```

Методика подобного рода работает только для конструкторов, определенных в такой старомодной манере. Классы, создаваемые с ключевым словом `class`, не разрешают вызывать свои конструкторы без `new`.

Еще одно критически важное отличие между примерами 9.1 и 9.2 связано со способом именования объекта прототипа. В первом примере прототипом был `range.methods`. Это удобное и описательное имя, но оно произвольно. Во втором примере прототипом является `Range.prototype`, и такое имя обязательно. Вызов конструктора `Range()` приводит к автоматическому использованию `Range.prototype` в качестве прототипа нового объекта `Range`.

В заключение также обратите внимание на то, что осталось неизменившимся в примерах 9.1 и 9.2: методы в обоих классах, представляющих диапазон, определяются и вызываются одинаково. Поскольку в примере 9.2 демонстрируется идиоматический способ создания классов в версиях JavaScript, предшествующих ES6, в объекте прототипа не применяется сокращенный синтаксис определения методов ES6, а методы описываются явно посредством ключевого слова `function`. Но, как видите, реализация методов в обоих примерах одинакова.

Важно отметить, что ни в одном из двух примеров класса диапазона при определении конструкторов или методов не используются стрелочные функции. Вспомните из подраздела 8.1.3, что функции, определенные подобным образом, не имеют свойства `prototype` и потому не могут применяться как конструкторы. Кроме того, стрелочные функции наследуют ключевое слово `this` от контекста, где они определены, а не устанавливают его на основе объекта, на котором вызываются. В итоге они оказываются бесполезными для методов, т.к. определяющей характеристикой методов является использование `this` для ссылки на экземпляр, в отношении которого они были вызваны.

К счастью, новый синтаксис классов ES6 не разрешает определять методы с помощью стрелочных функций, так что во время применения синтаксиса ES6 вы не совершите ошибку подобного рода. Вскоре мы раскроем ключевое слово `class` из ES6, но сначала нужно подробнее обсудить конструкторы.

9.2.1. Конструкторы, идентичность классов и операция `instanceof`

Как вы уже видели, объект прототипа считается основополагающим для идентичности класса: два объекта будут экземплярами одного и того же класса тогда и только тогда, когда они унаследованы от того же самого объекта прототипа. Функция конструктора, которая инициализирует состояние нового объекта, основополагающей не является: две функции конструктора могут иметь свойства `prototype`, указывающие на тот же самый объект прототипа. Тогда оба конструктора могут использоваться для создания экземпляров одного и того же класса.

Хотя конструкторы не настолько основополагающие как прототипы, конструктор служит публичным лицом класса. Наиболее очевидно то, что имя конструктора обычно принимается как имя класса. Например, мы говорим, что конструктор `Range()` создает объекты `Range`. Однако по существу конструкторы применяются в качестве правостороннего операнда операции `instanceof` при проверке объектов на предмет членства в классе. Если мы имеем объект `r` и хотим выяснить, является ли он объектом `Range`, то можем записать:

```
r instanceof Range          // => true: r наследуется от Range.prototype
```

Операция `instanceof` была описана в подразделе 4.9.4. Левосторонний операнд должен быть проверяемым объектом, а правосторонний — функцией конструктора, которая названа по имени класса. Выражение `o instanceof C` вычисляется как `true`, если объект `o` унаследован от `C.prototype`. Наследование не обязано быть прямым: если объект `o` унаследован от объекта, который унаследован от объекта, унаследованного от `C.prototype`, тогда выражение по-прежнему будет вычисляться как `true`.

Говоря формально, в предыдущем примере кода операция `instanceof` не проверяет, действительно ли объект `r` был инициализирован конструктором `Range`. Взамен она проверяет, унаследован ли объект `r` от `Range.prototype`. Если мы определим функцию `Strange()` и установим ее прототип таким же, как `Range.prototype`, тогда объекты, создаваемые с помощью `new Strange()`, будут считаться объектами `Range` с точки зрения операции `instanceof` (тем не менее, на самом деле они не будут работать как объекты `Range`, потому что их свойства `from` и `to` не инициализированы):

```
function Strange() {}
Strange.prototype = Range.prototype;
new Strange() instanceof Range    // => true
```

Хотя операция `instanceof` в действительности не может проверить факт использования конструктора, она все же задействует функцию конструктора

в своей правой стороне, т.к. конструкторы являются открытой идентичностью классов.

Если вы хотите проверить цепочку прототипов объекта на предмет наличия специфического прототипа и не хотите применять функцию конструктора в качестве посредника, тогда можете использовать метод `isPrototypeOf()`. Скажем, в примере 9.1 мы определяли класс без функции конструктора, а потому применить `instanceof` с таким классом не удастся. Однако взамен мы могли бы проверить, был ли объект `r` членом такого класса без конструктора, посредством следующего кода:

```
range.methods.isPrototypeOf(r);   // range.methods - объект прототипа
```

9.2.2. Свойство `constructor`

В примере 9.2 мы устанавливали `Range.prototype` в новый объект, который содержал методы для нашего класса. Несмотря на то что эти методы было удобно выражать как свойства одиночного объектного литерала, необходимость в создании нового объекта на самом деле отсутствовала. В качестве конструктора можно использовать любую функцию JavaScript (кроме стрелочных, генераторных и асинхронных функций), а для вызова функции как конструктора нужно лишь свойство `prototype`. По этой причине каждая обыкновенная функция JavaScript[1] автоматически располагает свойством `prototype`. Значением свойства `prototype` будет объект, который имеет единственное неперечислимое свойство `constructor`. Значением свойства `constructor` является объект функции:

```
let F = function() {};  // Объект функции
let p = F.prototype;    // Объект прототипа, ассоциированный с F
let c = p.constructor;  // Функция, ассоциированная с прототипом
c === F                 // => true: F.prototype.constructor === F
                        //          для любого объекта F
```

Существование предварительно определенного объекта прототипа с его свойством `constructor` означает, что объекты обычно наследуют свойство `constructor`, которое ссылается на их конструктор. Поскольку конструкторы служат открытой идентичностью класса, свойство `constructor` выдает класс объекта:

```
let o = new F();          // Создать объект o класса F
o.constructor === F       // => true: свойство constructor указывает класс
```

На рис. 9.1 иллюстрируется отношение между функцией конструктора, ее объектом прототипа, обратной ссылкой из прототипа на конструктор и экземплярами, созданными с помощью конструктора.

[1] Исключая функции, возвращаемые методом `Function.bind()` из ES5. Привязанные функции не имеют собственного свойства прототипа, а при вызове в качестве конструкторов используют прототип лежащей в основе функции.

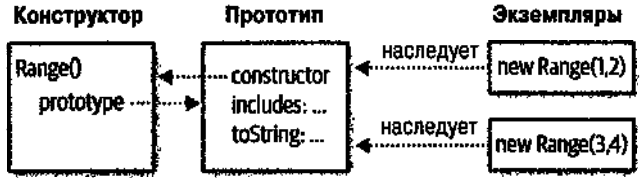


Рис. 9.1. Функция конструктора, ее прототип и экземпляры

Обратите внимание, что на рис. 9.1 в качестве примера применяется конструктор `Range()`. Тем не менее, в действительности класс `Range`, который был определен в примере 9.2, переопределяет предварительно определенный объект `Range.prototype` собственным объектом. И определенный им новый объект прототипа не имеет свойства `constructor`. Таким образом, экземпляры класса `Range` в том виде, в каком они определены, не имеют свойства `constructor`. Мы можем устранить проблему, явно добавляя конструктор к прототипу:

```
Range.prototype = {
  constructor: Range,//Явно установить обратную ссылку на конструктор
  /* далее идут определения методов */
};
```

Еще одна распространенная методика, которую вы, вероятно, встретите в более старом коде JavaScript, предусматривает использование предварительно определенного объекта прототипа с его свойством `constructor` и добавление методов к нему методов по одному за раз с применением кода следующего вида:

```
// Расширить предварительно определенный объект Range.prototype,
// чтобы не переопределять автоматически созданное
// свойство Range.prototype.constructor.
Range.prototype.includes = function(x) {
   return this.from <= x && x <= this.to;
};
Range.prototype.toString = function() {
   return "(" + this.from + "..." + this.to + ")";
};
```

9.3. Классы с ключевым словом `class`

Классы были частью JavaScript, начиная с самой первой версии языка, но с появлением в ES6 ключевого слова `class` они наконец-то получили собственный синтаксис. В примере 9.3 показано, как выглядит наш класс `Range`, когда он записывается с использованием нового синтаксиса.

Пример 9.3. Класс `Range`, переписанный с применением ключевого слова `class`

```
class Range {
   constructor(from, to) {
      // Сохранить начальную и конечную точки (состояние)
      // нового объекта диапазона.
```

```
        // Эти свойства не являются унаследованными и они уникальны
        // для этого объекта.
        this.from = from;
        this.to = to;
    }
    // Возвращает true, если x входит в диапазон, и false в противном
    // случае. Метод работает как с числовыми, так и с текстовыми
    // диапазонами и диапазонами Date.
    includes(x) { return this.from <= x && x <= this.to; }

    // Генераторная функция, которая делает экземпляры класса итерируемыми.
    // Обратите внимание, что она работает только с числовыми диапазонами.
    *[Symbol.iterator]() {
        for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
    }
    // Возвращает строковое представление диапазона.
    toString() { return `(${this.from}...${this.to})`; }
}
// Пример использования нового класса Range.
let r = new Range(1,3);    // Создать объект Range
r.includes(2)  // => true: 2 входит в диапазон
r.toString()   // => "(1...3)"
[...r]         // => [1, 2, 3]; преобразовать в массив через итератор
```

Важно понять, что классы, определенные в примерах 9.2 и 9.3, работают совершенно одинаково. Введение в язык ключевого слова `class` не изменило фундаментальную природу основанных на прототипах классов JavaScript. И хотя в примере 9.3 используется ключевое слово `class`, результирующий объект `Range` является функцией конструктора — точно как в версии, определенной в примере 9.2. Новый синтаксис `class` отличается ясностью и удобством, но лучше воспринимать его как "синтаксический сахар" по отношению к более фундаментальному механизму определения классов, показанному в примере 9.2.

Полезно отметить несколько моментов, касающихся синтаксиса `class` в примере 9.3.

- Класс объявляется с помощью ключевого слова `class`, за которым следует имя класса и тело класса в фигурных скобках.
- Тело класса содержит определения методов, которые применяют сокращенное определение методов в объектном литерале (он также использовался в примере 9.1), где ключевое слово `function` опущено. Однако в отличие от объектных литералов, для отделения методов друг от друга запятые не применяются. (Хотя тело класса внешне похоже на объектный литерал, они представляют собой разные вещи. В частности, определение свойств посредством пар имя/значение не поддерживается.)
- Ключевое слово `constructor` используется для определения функции конструктора класса. Тем не менее, определяемая функция на самом деле не называется "constructor". Оператор объявления `class` определяет новую переменную `Range` и присваивает ей значение этой специальной функции `constructor`.

- Если ваш класс не нуждается в инициализации, тогда можете опустить ключевое слово `constructor` вместе с его телом, и неявно будет создана пустая функция конструктора.

Если вы хотите определить класс, который является подклассом другого класса, или *унаследованным от* него, то можете применить вместе с `class` ключевое слово `extends`:

```
// Класс Span подобен Range, но вместо инициализации значениями
// начала и конца мы инициализируем его значениями начала и длины.
class Span extends Range {
    constructor(start, length) {
        if (length >= 0) {
            super(start, start + length);
        } else {
            super(start + length, start);
        }
    }
}
```

Создание подклассов — отдельная тема. Мы возвратимся к ней в разделе 9.5 и обсудим ключевые слова `extends` и `super`.

Подобно объявлениям функций объявления классов имеют формы в виде оператора и выражения. Точно так же, как можно записать:

```
let square = function(x) { return x * x; };
square(3)                 // => 9
```

также допускается записать следующим образом:

```
let Square = class { constructor(x) { this.area = x * x; } };
new Square(3).area        // => 9
```

Как и выражения определения функций, выражения определения классов могут включать необязательное имя класса. В случае предоставления такого имени оно будет определено только внутри самого тела класса.

Наряду с тем, что выражения определения функций довольно распространены (особенно сокращения в виде стрелочных функций), выражения определения классов при программировании на JavaScript используются нечасто, если только вы не пишете функцию, которая принимает класс в качестве аргумента и возвращает подкласс.

Мы завершим представление ключевого слова `class` упоминанием пары важных моментов, о которых следует знать и которые не являются очевидными в синтаксисе `class`.

- Весь код в теле объявления `class` неявно подчиняется требованиям строгого режима (см. подраздел 5.6.3), даже если отсутствует директива `"use strict"`. Таким образом, в теле класса нельзя применять, например, восьмеричные целочисленные литералы или оператор `with`, и если вы забудете объявить переменную перед ее использованием, то с более высокой вероятностью получите ошибку.

- В отличие от объявлений функций объявления классов не "поднимаются". Вспомните из подраздела 8.1.1, что определения функций ведут себя так, словно они перемещены в начало включающего файла или функции, т.е. функцию можно вызывать в коде, находящемся до ее фактического определения. Невзирая на некоторые сходства объявлений классов и функций, они не разделяют поведение подъема: вы не можете создать экземпляр класса до его объявления.

9.3.1. Статические методы

Определить статический метод внутри тела класса можно, снабдив объявление метода префиксом в виде ключевого слова `static`. Статические методы определяются как свойства функции конструктора, а не свойства объекта прототипа.

Предположим, что мы добавили в пример 9.3 следующий код:

```
static parse(s) {
    let matches = s.match(/^\((\d+)\.\.\.(\d+)\)$/);
    if (!matches) {
        // Не удается разобрать диапазон
        throw new TypeError(`Cannot parse Range from "${s}".`)
    }
    return new Range(parseInt(matches[1]), parseInt(matches[2]));
}
```

В вышеприведенном коде определяется метод `Range.parse()`, не `Range.prototype.parse()`, который должен вызываться через конструктор, а не через экземпляр:

```
let r = Range.parse('(1...10)');     // Возвращается новый объект Range
r.parse('(1...10)');                 // TypeError: r.parse - не функция
```

Иногда статические методы называют *методами класса* из-за того, что они вызываются с применением имени класса/конструктора. Такой термин используется как противоположность обыкновенным *методам экземпляра*, которые вызываются на экземплярах класса. Поскольку статические методы вызываются на конструкторе, а не на индивидуальном экземпляре, в них почти никогда не имеет смысла применять ключевое слово `this`.

Статические методы будут добавлены в пример 9.4.

9.3.2. Методы получения, установки и других видов

В теле класса можно определять методы получения и установки (см. подраздел 6.10.6), как это делалось в объектных литералах. Единственное отличие в том, что в теле класса не нужно помещать запятую после метода получения или установки. В примере 9.4 демонстрируется практическая версия метода получения в классе.

В целом весь сокращенный синтаксис определения методов, разрешенный в объектных литералах, также является допустимым в телах классов, включая генераторные методы (помечаемые посредством *) и методы, имена которых

представлены как значения выражений в квадратных скобках. В действительности вы уже видели (в примере 9.3) генераторный метод с вычисляемым именем, который делает класс Range итерируемым:

```
*[Symbol.iterator]() {
   for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
}
```

9.3.3. Открытые, закрытые и статические поля

При обсуждении классов, определяемых с помощью ключевого слова class, мы затронули только определение методов внутри тела класса. Стандарт ES6 разрешает создавать только методы (включая методы получения, установки и генераторы) и статические методы; синтаксис для определения полей отсутствует. Если вы хотите определить поле (которое представляет собой лишь синоним "свойства" в объектно-ориентированном программировании) в экземпляре класса, то должны делать это в функции конструктора или в одном из методов. А если вас интересует определение статического поля для класса, тогда вы должны делать это вне тела класса после того, как класс был определен. В примере 9.4 присутствуют поля обоих видов.

Однако стандартизация для расширенного синтаксиса, который делает возможным определение полей экземпляра и статических полей в открытой и закрытой форме, уже ведется. По состоянию на начало 2020 года код, показанный в оставшейся части раздела, пока еще не является стандартом JavaScript, но уже поддерживается в веб-браузере Chrome и частично (только открытые поля экземпляра) в Firefox. Синтаксис открытых полей экземпляра часто используется программистами на JavaScript, работающими с фреймворком React и транспилятором Babel.

Предположим, что вы написали класс следующего вида с конструктором, который инициализирует три поля:

```
class Buffer {
   constructor() {
      this.size = 0;
      this.capacity = 4096;
      this.buffer = new Uint8Array(this.capacity);
   }
}
```

С помощью нового синтаксиса для полей экземпляра, который вероятно будет стандартизирован, вы могли бы взамен записать так:

```
class Buffer {
   size = 0;
   capacity = 4096;
   buffer = new Uint8Array(this.capacity);
}
```

Код инициализации полей был вынесен из конструктора и теперь находится прямо в теле класса. (Разумеется, этот код по-прежнему будет выполняться

как часть конструктора. Если вы не определили конструктор, тогда поля инициализируются как часть неявно созданного конструктора.) Префиксы `this.` с левой стороны операций присваивания исчезли, но имейте в виду, что `this.` все равно придется применять для ссылки на поля даже с правой стороны инициализирующих операций присваивания. Преимущество инициализации полей экземпляра подобным образом заключается в том, что такой синтаксис разрешает (но не требует) помещать инициализаторы в начало определения класса, позволяя читателям четко видеть, какие поля будут хранить состояние каждого экземпляра. Вы можете объявлять поля без инициализаторов, просто указывая имя поля, за которым следует точка с запятой. В этом случае начальным значением поля будет `undefined`. Наилучший стиль программирования предусматривает обеспечение явных начальных значений для всех полей класса.

До добавления такого синтаксиса полей тело класса во многом напоминало объектный литерал, в котором используется сокращенный синтаксис определения методов, но без запятых. Синтаксис полей со знаками равенства и точками с запятой вместо двоеточий и запятых проясняет тот факт, что тело класса — не совсем то же, что и объектный литерал.

В том же самом предложении, направленном на стандартизацию полей экземпляра, также определяются закрытые поля экземпляра. Если вы применяете синтаксис инициализации полей, показанный в предыдущем примере, для определения поля, имя которого начинается с # (обычно недопустимый символ в идентификаторах JavaScript), то такое поле будет пригодным к употреблению (с префиксом #) внутри тела класса, но невидимым и недоступным (а потому неизменяемым) любому коду за рамками тела класса. Если для предшествующего гипотетического класса `Buffer` вы хотите гарантировать, что пользователи не сумеют случайно модифицировать поле `size` экземпляра, тогда могли бы взамен использовать закрытое поле `#size` и определить функцию получения, чтобы предоставить доступ только для чтения к значению:

```
class Buffer {
    #size = 0;
    get size() { return this.#size; }
}
```

Обратите внимание, что закрытые поля должны объявляться с применением нового синтаксиса полей до того, как их можно будет использовать. Вы не можете просто записать `this.#size = 0;` в конструкторе класса, если только не включили "объявление" поля прямо в тело класса.

Наконец, связанное предложение направлено на стандартизацию применения ключевого слова `static` для полей. Если вы добавите `static` перед объявлениями открытых или закрытых полей, то такие поля будут создаваться как свойства функции конструктора, а не свойства экземпляров. Рассмотрим ранее определенный нами статический метод `Range.parse()`. Он содержит довольно сложное регулярное выражение, которое заслуживает того, чтобы быть вынесенным в собственное статическое поле. С помощью предложенного нового синтаксиса статических полей мы могли бы сделать это примерно так:

```
  static integerRangePattern = /^\((\d+)\.\.\.(\d+)\)$/;
  static parse(s) {
    let matches = s.match(Range.integerRangePattern);
    if (!matches) {
      throw new TypeError(`Cannot parse Range from "${s}".`)
    }
    return new Range(parseInt(matches[1]), matches[2]);
  }
```

Если желательно, чтобы это статическое поле было доступным только внутри класса, тогда мы можем сделать его закрытым с использованием имени наподобие #pattern.

9.3.4. Пример: класс для представления комплексных чисел

В примере 9.4 определяется класс для представления комплексных чисел. Он относительно прост, но содержит методы экземпляра (включая методы получения), статические методы, поля экземпляра и статические поля. В закомментированном коде демонстрируется, как можно было бы применять пока еще не ставший стандартным синтаксис для определения полей экземпляра и статических полей внутри тела класса.

Пример 9.4. `Complex.js`: класс для представления комплексных чисел

```
/**
 * Экземпляры данного класса Complex представляют комплексные числа.
 * Вспомните, что комплексное число - это сумма вещественного числа и мнимого
 * числа, а мнимое число i представляет собой квадратный корень из -1.
 */
class Complex {
  // После стандартизации объявлений полей класса мы могли бы здесь объявить
  // закрытые поля для хранения вещественной и мнимой частей комплексного
  // числа посредством кода следующего вида:
  // #r = 0;
  // #i = 0;
  // Функция конструктора определяет поля экземпляра r и i для каждого
  // создаваемого ею экземпляра. Эти поля хранят вещественную и мнимую
  // части комплексного числа: они являются состоянием объекта.
  constructor(real, imaginary) {
    this.r = real; // Поле, хранящее вещественную часть комплексного числа
    this.i = imaginary;  // Поле, хранящее мнимую часть комплексного числа
  }
  // Два метода экземпляра для сложения и умножения
  // комплексных чисел. Если c и d - экземпляры данного класса,
  // то мы можем записывать c.plus(d) или d.times(c).
  plus(that) {
    return new Complex(this.r + that.r, this.i + that.i);
  }
  times(that) {
    return new Complex(this.r * that.r - this.i * that.i,
                       this.r * that.i + this.i * that.r);
  }
```

```
    // Статические версии методов для действий с комплексными числами.
    // Мы можем записывать Complex.sum(c,d) и Complex.product(c,d).
    static sum(c, d) { return c.plus(d); }
    static product(c, d) { return c.times(d); }

    // Несколько методов экземпляра, которые определены как методы получения,
    // так что они используются подобно полям. Методы получения вещественной
    // и мнимой частей были бы полезными в случае применения закрытых
    // полей this.#r и this.#i.
    get real() { return this.r; }
    get imaginary() { return this.i; }
    get magnitude() { return Math.hypot(this.r, this.i); }

    // Классы почти всегда должны иметь метод toString().
    toString() { return `{${this.r},${this.i}}`; }

    // Часто полезно определять метод для проверки, представляют
    // ли два экземпляра класса одно и то же значение.
    equals(that) {
        return that instanceof Complex &&
            this.r === that.r &&
            this.i === that.i;
    }

    // После того, как статические поля станут поддерживаться внутри тела класса,
    // мы сможем определить полезную константу Complex.ZERO:
    // static ZERO = new Complex(0,0);
}

// Ниже приведены поля класса, хранящие полезные предварительно
// определенные комплексные числа.
Complex.ZERO = new Complex(0,0);
Complex.ONE = new Complex(1,0);
Complex.I = new Complex(0,1);
```

После определения класса Complex в примере 9.4 мы можем использовать конструктор, поля экземпляра, методы экземпляра, поля класса и методы класса, как показано ниже:

```
let c = new Complex(2, 3);       // Создать новый объект с помощью
                                 // конструктора
let d = new Complex(c.i, c.r);   // Использовать поля экземпляра с
c.plus(d).toString()     // => "{5,5}"; использовать методы экземпляра
c.magnitude    // => Math.hypot(2,3); использовать функцию получения
Complex.product(c, d)    // => new Complex(0, 13); статический метод
Complex.ZERO.toString() // => "{0,0}"; статическое свойство
```

9.4. Добавление методов в существующие классы

Механизм наследования на основе прототипов JavaScript является динамическим: объект наследует свойства от своего прототипа, даже если свойства прототипа изменяются после создания объекта. Это означает, что мы можем

дополнять классы JavaScript, просто добавляя новые методы к их объектам прототипов.

Скажем, ниже приведен код, который добавляет в класс Complex из примера 9.4 метод для вычисления комплексно-сопряженного числа:

```
// Возвращает комплексное число, которое является
// комплексно-сопряженным для данного числа.
Complex.prototype.conj = function() { return new Complex(this.r, -this.i); };
```

Объект прототипа встроенных классов JavaScript тоже открыт в аналогичной манере, т.е. мы можем добавлять методы к числам, строкам, массивам, функциям и т.д. Поступать так удобно для реализации новых языковых средств в более старых версиях языка:

```
// Если новый метод startsWith() класса String еще не определен...
if (!String.prototype.startsWith) {
   // ...тогда определить его с использованием более старого
   // метода indexOf().
   String.prototype.startsWith = function(s) {
      return this.indexOf(s) === 0;
   };
}
```

Вот еще один пример:

```
// Вызвать функцию f много раз, передавая количество итераций.
// Например, для трехкратного вывода "hello":
// let n = 3;
// n.times(i => { console.log(`hello ${i}`); });
Number.prototype.times = function(f, context) {
   let n = this.valueOf();
   for(let i = 0; i < n; i++) f.call(context, i);
};
```

Подобное добавление методов к прототипам встроенных типов обычно считается плохой идеей, поскольку приводит к путанице и проблемам с совместимостью в будущем, если в новой версии JavaScript появится метод с тем же самым именем. Допускается даже добавлять методы в `Object.prototype`, делая их доступными для всех объектов. Но такое действие никогда не заканчивается чем-то хорошим, потому что свойства, добавленные к `Object.prototype`, видимы циклам `for/in` (хотя вы можете избежать этого за счет применения метода `Object.defineProperty()`, описанного в разделе 14.1, чтобы сделать новое свойство неперечислимым).

9.5. Подклассы

В объектно-ориентированном программировании класс B может *расширять* или *быть подклассом* класса A. Мы говорим, что A является *суперклассом*, а B — *подклассом*. Экземпляры B наследуют методы от A. Класс B может определять собственные методы, часть которых может *переопределять* методы с такими же именами, определенные классом A. Когда какой-то метод класса B переопреде-

ляет метод класса A, переопределяющий метод в B часто нуждается в вызове переопределенного метода из A. Аналогично конструктор подкласса B() обычно обязан вызывать конструктор суперкласса A() для обеспечения полноценной инициализации экземпляров.

Текущий раздел начинается с демонстрации того, как определять подклассы старым способом, принятым до выхода ES6, после чего объясняется, каким образом создавать подклассы с использованием ключевых слов `class` и `extends` и вызывать методы конструкторов суперклассов посредством ключевого слова `super`. В следующем подразделе речь идет об отказе от суперклассов и применении композиции объектов вместо наследования. Раздел заканчивается расширенным примером, в котором будет определена иерархия классов `Set` и показано, как можно использовать абстрактные классы для отделения интерфейса от реализации.

9.5.1. Подклассы и прототипы

Предположим, что нужно определить подкласс Span класса Range из примера 9.2. Подкласс Span будет работать в точности как Range, но вместо инициализации значениями начала и конца диапазона мы будем указывать значения начала и расстояния, или промежуток. Экземпляр класса Span также является экземпляром суперкласса Range. Экземпляр Span наследует настроенный метод `toString()` от `Span.prototype`, но чтобы быть подклассом Range, он обязан также наследовать методы (вроде `includes()`) от `Range.prototype`.

Пример 9.5. `Span.js`: простой подкласс класса `Range`

```
// Функция конструктора для нашего подкласса.
function Span(start, span) {
    if (span >= 0) {
        this.from = start;
        this.to = start + span;
    } else {
        this.to = start;
        this.from = start + span;
    }
}

// Обеспечить, чтобы прототип Span наследовался от прототипа Range.
Span.prototype = Object.create(Range.prototype);

// Мы не хотим наследовать Range.prototype.constructor, поэтому
// определяем собственное свойство constructor.
Span.prototype.constructor = Span;

// За счет определения собственного метода toString() подкласс Span
// переопределяет метод toString(), который иначе он унаследовал бы от Range.
Span.prototype.toString = function() {
    return `(${this.from}... +${this.to - this.from})`;
};
```

Чтобы сделать Span подклассом Range, нам нужно организовать наследование Span.prototype от Range.prototype. Ниже приведена ключевая строка кода из предыдущего примера; если ее смысл для вас ясен, тогда вы поймете, как работают подклассы в JavaScript:

```
Span.prototype = Object.create(Range.prototype);
```

Объекты, создаваемые с помощью конструктора Span(), будут унаследованными от объекта Span.prototype. Но мы создали этот объект для наследования от Range.prototype, так что объекты Span будут наследоваться и от Span.prototype, и от Range.prototype.

Вы можете заметить, что конструктор Span() устанавливает те же самые свойства from и to, как и конструктор Range(), а потому для инициализации нового объекта вызывать конструктор Range() не требуется. Аналогично метод toString() класса Span полностью заново реализует преобразование в строку без необходимости в вызове версии toString() из Range. В итоге Span становится особым случаем, и мы действительно можем избежать создания подклассов подобного рода лишь потому, что знаем детали реализации суперкласса. Надежный механизм создания подклассов должен предоставлять классам возможность вызова методов и конструктора своего суперкласса, но до выхода ES6 в языке JavaScript отсутствовал простой способ выполнять такие действия.

К счастью, в ES6 проблемы были решены посредством ключевого слова super как части синтаксиса class. В следующем подразделе будет показано, каким образом оно работает.

9.5.2. Создание подклассов с использованием extends и super

В ES6 и последующих версиях появилась возможность указывать класс в качестве суперкласса, просто добавляя к объявлению класса конструкцию extends, и это допускается делать даже для встроенных классов:

```
// Тривиальный подкласс Array, который добавляет методы
// получения для первого и последнего элементов.
class EZArray extends Array {
  get first() { return this[0]; }
  get last() { return this[this.length-1]; }
}

let a = new EZArray();
a instanceof EZArray    // => true: a - экземпляр подкласса
a instanceof Array      // => true: a - также экземпляр суперкласса
a.push(1,2,3,4);        // a.length == 4; можно использовать
                        //                унаследованные методы
a.pop()                 // => 4: еще один унаследованный метод
a.first                 // => 1: метод получения первого элемента,
                        //       определенный в подклассе
a.last                  // => 3: метод получения последнего элемента,
                        //       определенный в подклассе
```

```
a[1]                  // => 2: обычный синтаксис доступа к элементам
                      //       массива по-прежнему работает
Array.isArray(a)      // => true: экземпляр подкласса действительно
                      //           является массивом
EZArray.isArray(a) // => true: подкласс наследует и статические методы!
```

В подклассе `EZArray` определены два простых метода получения. Экземпляры `EZArray` ведут себя как обыкновенные массивы, и мы можем применять методы и свойства вроде `push()`, `pop()` и `length`. Но мы также можем использовать методы получения `first` и `last`, которые определены в подклассе. Наследуются не только методы экземпляра наподобие `pop()`, но и статические методы, такие как `Array.isArray`. Это новая возможность синтаксиса определения классов ES6: `EZArray()` является функцией, но она наследуется от `Array()`:

```
// EZArray наследует методы экземпляра, поскольку EZArray.prototype
// наследуется от Array.prototype.
Array.prototype.isPrototypeOf(EZArray.prototype)  // => true

// Вдобавок EZArray наследует статические методы и свойства, потому что
// EZArray наследуется от Array. Это специальная возможность ключевого
// слова extends, которая не была доступна до выхода ES6.
Array.isPrototypeOf(EZArray)  // => true
```

Наш подкласс `EZArray` слишком прост, чтобы многому научить. Пример 9.6 более развит; в нем определяется подкласс `TypedMap` встроенного класса `Map`, который добавляет проверку того, что ключи и значения отображения имеют указанные типы (согласно `typeof`). А еще важнее то, что пример демонстрирует применение ключевого слова `super` для вызова конструктора и методов суперкласса.

Пример 9.6. `TypedMap.js`: подкласс `Map`, который проверяет типы ключей и значений

```
class TypedMap extends Map {
   constructor(keyType, valueType, entries) {
      // Если записи entries определены, тогда проверить их типы.
      if (entries) {
         for(let [k, v] of entries) {
            if (typeof k !== keyType || typeof v !== valueType) {
              throw new TypeError(`Неправильный тип для записи [${k}, ${v}]`);
            }
         }
      }

      //Инициализировать суперкласс начальными записями (с проверенными типами)
      super(entries);

      // Затем инициализировать подкласс this, сохранив типы.
      this.keyType = keyType;
      this.valueType = valueType;
   }

   // Теперь переопределить метод set(), добавив проверку типов
   // для любых новых записей, добавляемых к отображению.
```

```
  set(key, value) {
    // Сгенерировать ошибку, если ключ или значение имеет неправильный тип.
    if (this.keyType && typeof key !== this.keyType) {
      throw new TypeError(`${key} не относится к типу ${this.keyType}`);
    }
    if (this.valueType && typeof value !== this.valueType) {
      throw new TypeError(`${value} не относится к типу ${this.valueType}`);
    }
    // Если типы корректны, тогда мы вызывает версию метода set()
    // из суперкласса для фактического добавления записи в отображение.
    // Мы возвращаем то, что возвращает метод суперкласса.
    return super.set(key, value);
  }
}
```

Первыми двумя аргументами конструктора `TypedMap()` являются желательные типы ключей и значений. Они должны быть строками наподобие `"number"` и `"boolean"`, которые возвращает операция `typeof`. Можно также передать третий аргумент: массив (или любой итерируемый объект) массивов `[ключ,значение]`, который задает начальные записи в отображении. В случае указания начальных записей конструктор первым делом выполнит проверку на предмет корректности их типов. Затем конструктор вызывает конструктор суперкласса, используя ключевое слово `super`, как если бы оно было именем функции. Конструктор `Map()` принимает один необязательный аргумент: итерируемый объект с массивами `[ключ,значение]`. Таким образом, необязательный третий аргумент конструктора `TypedMap()` является необязательным первым аргументом конструктора `Map()`, который мы передаем конструктору суперкласса с помощью `super(entries)`.

После вызова конструктора суперкласса для инициализации состояния суперкласса конструктор `TypedMap()` далее инициализирует собственное состояние подкласса, устанавливая `this.keyType` и `this.valueType` в указанные типы. Ему необходимо установить эти свойства, чтобы их можно было снова применять в методе `set()`.

Существует несколько важных правил, которые нужно знать об использовании `super()` в конструкторах.

- Если вы определяете класс с ключевым словом `extends`, тогда в конструкторе вашего класса должно применяться `super()` для вызова конструктора суперкласса.
- Если вы не определили конструктор в своем подклассе, то он будет определен автоматически. Такой неявно определенный конструктор просто передает `super()` любые предоставленные значения.
- Вы не можете использовать ключевое слово `this` в своем конструкторе, пока не будет вызван конструктор суперкласса посредством `super()`. Это обеспечивает соблюдение правила о том, что суперклассы должны инициализировать себя перед подклассами.

- Специальное выражение `new.target` не определено в функциях, которые вызываются без ключевого слова `new`. Тем не менее, в функциях конструкторов `new.target` является ссылкой на вызванный конструктор. Когда конструктор подкласса вызывается и применяет `super()` для вызова конструктора суперкласса, то конструктор суперкласса будет видеть конструктор подкласса как значение `new.target`. Хорошо спроектированный суперкласс не должен знать, создаются ли из него подклассы, но `new.target.name` иногда полезно использовать, например, в журнальных сообщениях.

После конструктора в примере 9.6 следует метод по имени `set()`. В суперклассе `Map` определен метод `set()` для добавления новой записи в отображение. Мы говорим, что метод `set()` в `TypedMap` *переопределяет* метод `set()` своего суперкласса. Нашему простому подклассу `TypedMap` ничего не известно о добавлении новых записей в отображение, но он знает, как проверять типы, так что именно это он сначала и делает, проверяя корректность типов добавляемых к отображению ключа и значения и генерируя ошибку в случае неправильных типов. Сам метод `set()` подкласса не имеет никакой возможности добавить ключ и значение к отображению, но для такой цели предназначен метод `set()` суперкласса. Таким образом, мы снова применяем ключевое слово `super` для вызова версии метода `set()` из суперкласса. В данном контексте `super` работает подобно ключевому слову `this`: оно относится к текущему объекту, но делает возможным доступ к переопределенным методам суперкласса.

В конструкторах вы обязаны вызывать конструктор суперкласса, прежде чем можно будет получить доступ к `this` и самостоятельно инициализировать новый объект. При переопределении метода таких правил нет. Вызывать метод суперкласса внутри метода, который его переопределяет, вовсе не обязательно. В случае использования `super` для вызова переопределенного метода (или любого метода) из суперкласса это можно делать в начале, в середине или в конце переопределяющего метода.

Наконец, прежде чем мы завершим рассмотрение примера `TypedMap`, нелишне отметить, что этот класс является идеальным кандидатом для применения закрытых полей. В текущей реализации класса пользователь может изменить свойство `keyType` или `valueType`, чтобы обойти проверку типов. Как только будут поддерживаться закрытые поля, мы сможем изменить имеющиеся свойства на `#keyType` и `#valueType`, чтобы их нельзя было модифицировать извне.

9.5.3. Делегирование вместо наследования

Ключевое слово `extends` облегчает создание подклассов. Но это совершенно не означает, что вы *должны* создавать много подклассов. Если вы хотите написать класс, который разделяет поведение с каким-то другим классом, то можете попытаться унаследовать такое поведение путем создания подкласса. Но временами получить желаемое поведение в своем классе легче и гибче за счет создания экземпляра другого класса и по мере необходимости делегирования

ему соответствующей работы. Вы строите новый класс, не создавая подкласс, а организуя оболочку для других классов или "компонуя" их. Такой подход часто называют "композицией" и он является часто цитируемым принципом объектно-ориентированного программирования, согласно которому нужно "отдавать предпочтение композиции перед наследованием"[2].

Пусть, например, нас интересует класс `Histogram`, представляющий гистограмму, который ведет себя кое в чем похоже на класс `Set` из JavaScript, но вместо простого отслеживания, было ли значение добавлено к множеству, он поддерживает счетчик, который показывает, сколько раз значение добавлялось. Поскольку API-интерфейс для класса `Histogram` подобен API-интерфейсу `Set`, мы могли бы рассмотреть вариант создания подкласса `Set` и добавления метода `count()`. С другой стороны, начав думать о реализации метода `count()`, мы могли бы осознать, что класс `Histogram` больше похож на `Map`, чем на `Set`, т.к. ему нужно поддерживать отображение между значениями и количеством их добавлений. Таким образом, вместо создания подкласса `Set` мы можем создать класс, который определяет API-интерфейс, подобный `Set`, но реализует его методы путем делегирования работы внутреннему объекту `Map`. В примере 9.7 показано, как можно было бы это сделать.

Пример 9.7. `Histogram.js`: класс, подобный `Set`, который реализован с помощью делегирования

```
/**
 * Класс, подобный Set, который отслеживает, сколько раз добавлялось значение.
 * Вызывайте методы add() и remove(), как делали бы для Set, и вызывайте
 * count(), чтобы выяснить, сколько раз добавлялось заданное значение.
 * Стандартный итератор выдает значения, которые были добавлены хотя бы раз.
 * Используйте entries(), если хотите проходить по парам [значение, счетчик].
 */
class Histogram {
  // Для инициализации мы просто создаем объект Map,
  // чтобы делегировать ему выполнение работы.
  constructor() { this.map = new Map(); }

  // Для любого заданного ключа счетчик - это значение
  // в объекте Map или ноль, если ключ в Map отсутствует.
  count(key) { return this.map.get(key) || 0; }

  // Метод has() как в классе Set возвращает true в случае ненулевого счетчика.
  has(key) { return this.count(key) > 0; }

  // Размер гистограммы - это просто количество записей в Map.
  get size() { return this.map.size; }

  // Чтобы добавить ключ, необходимо лишь инкрементировать его счетчик в Map.
  add(key) { this.map.set(key, this.count(key) + 1); }

  // Удаление ключа чуть сложнее, т.к. мы должны удалять
  // ключ из Map, если счетчик становится равным нулю.
```

[2] См. книгу Эриха Гамма и др. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* (изд. "Питер") или Джошуа Блоха *Java. Эффективное программирование*, 3-е изд. (пер. с англ, изд. "Диалектика", 2019 г.).

```
    delete(key) {
        let count = this.count(key);
        if (count === 1) {
            this.map.delete(key);
        } else if (count > 1) {
            this.map.set(key, count - 1);
        }
    }
    // При проходе по объекту Histogram просто возвращаются хранящиеся
    // в нем ключи.
    [Symbol.iterator]() { return this.map.keys(); }
    // Остальные методы итераторов всего лишь делегируют работу объекту Map.
    keys() { return this.map.keys(); }
    values() { return this.map.values(); }
    entries() { return this.map.entries(); }
}
```

Все, что делает конструктор `Histogram()` в примере 9.7 — создает объект `Map`. И большинство методов состоят из одной строки, которая просто делегирует работу методу объекта `Map`, делая реализацию довольно простой. Поскольку мы применяем делегирование, а не наследование, объект `Histogram` не является экземпляром `Set` или `Map`. Однако класс `Histogram` реализует несколько часто используемых методов `Set`, и в нетипизированных языках вроде JavaScript этого оказывается вполне достаточно: формальное отношение наследования временами удачно, но часто необязательно.

9.5.4. Иерархии классов и абстрактные классы

В примере 9.6 демонстрировалось создание подкласса `Map`, а в примере 9.7 — то, как взамен можно делегировать работу объекту `Map`, фактически не создавая подкласс чего-либо. Применение классов JavaScript для инкапсуляции данных и модульной организации кода часто является великолепным приемом, и вы можете обнаружить, что регулярно используете ключевое слово `class`. Но может выясниться, что вы отдаете предпочтение композиции перед наследованием и редко нуждаетесь в применении `extends` (кроме случаев, когда вы используете библиотеку или фреймворк, который требует расширения своих базовых классов).

Тем не менее, существуют обстоятельства, когда уместно множество уровней создания подклассов, и в конце главы будет приведен расширенный пример построения иерархии классов, которые представляют различные виды множеств. (Классы множеств, определенные в примере 9.8, похожи, но не полностью совместимы со встроенным классом `Set` в JavaScript.)

В примере 9.8 определено много подклассов, но также показано, как можно определять *абстрактный класс*, т.е. класс, не включающий полную реализацию, который послужит общим суперклассом для группы связанных подклассов. Абстрактный суперкласс может определять частичную реализацию, которую наследуют и разделяют все подклассы. Затем подклассам останется лишь оп-

ределить собственное уникальное поведение, реализовав абстрактные методы, которые определены, но не реализованы суперклассом. Обратите внимание, что в JavaScript отсутствует формальное определение абстрактных методов или абстрактных классов; я просто применяю здесь такое название для нереализованных методов и классов, реализованных не полностью.

Пример 9.8 хорошо прокомментирован и самодостаточен. Я рекомендую воспринимать его как заключительный пример в этой главе, посвященной классам. Финальный класс в примере 9.8 выполняет много манипуляций с битами посредством операций &, | и ~, которые были описаны в подразделе 4.8.3.

Пример 9.8. `Sets.js`: иерархия абстрактных и конкретных классов множеств

```
/**
 * Класс AbstractSet определяет единственный абстрактный метод has().
 */
class AbstractSet {
   // Сгенерировать ошибку, чтобы заставить подклассы определять
   // собственную работающую версию этого метода.
   has(x) { throw new Error("Абстрактный метод"); }
}
/**
 * NotSet - конкретный подкласс AbstractSet.
 * Все члены этого множества являются значениями, которые не будут членами
 * какого-то другого множества. Из-за определения в терминах другого множества
 * класс не допускает запись, а из-за наличия бесконечного количества членов он
 * не поддерживает перечисление. Все, что мы можем с ним делать - проверять на
 * предмет членства и преобразовывать в строку, использующую математическую запись.
 */
class NotSet extends AbstractSet {
   constructor(set) {
      super();
      this.set = set;
   }
   // Наша реализация унаследованного абстрактного метода.
   has(x) { return !this.set.has(x); }
   // И мы также переопределяем следующий метод из Object.
   toString() { return `{ x| x ∉ ${this.set.toString()} }`; }
}
/**
 * RangeSet - конкретный подкласс AbstractSet. Все его члены являются
 * значениями, которые находятся между границами from и to включительно.
 * Поскольку его членами могут быть числа с плавающей точкой, он не
 * поддерживает перечисление и не имеет значащего размера.
 */
class RangeSet extends AbstractSet {
   constructor(from, to) {
      super();
      this.from = from;
      this.to = to;
   }
```

```
    has(x) { return x >= this.from && x <= this.to; }
    toString() { return `{ x| ${this.from} ≤ x ≤ ${this.to} }`; }
}

/*
 * AbstractEnumerableSet - абстрактный подкласс AbstractSet. Он определяет
 * абстрактный метод получения, который возвращает размер множества, а также
 * определяет абстрактный итератор. Затем он реализует конкретные методы
 * isEmpty(), toString() и equals(). Подклассы, которые реализуют итератор,
 * метод получения размера и метод has(), свободно получают эти конкретные
 * методы.
 */
class AbstractEnumerableSet extends AbstractSet {
    get size() { throw new Error("Абстрактный метод"); }
    [Symbol.iterator]() { throw new Error("Абстрактный метод"); }

    isEmpty() { return this.size === 0; }
    toString() { return `{${Array.from(this).join(", ")}}`; }
    equals(set) {
        // Если другое множество не является AbstractEnumerableSet,
        // то оно не равно этому множеству.
        if (!(set instanceof AbstractEnumerableSet)) return false;

        // Если множества не имеют одинаковые размеры, то они не равны.
        if (this.size !== set.size) return false;

        // Проход в цикле по элементам этого множества.
        for(let element of this) {
            // Если элемент не находится в другом множестве, то они не равны.
            if (!set.has(element)) return false;
        }

        // Элементы совпадают, поэтому множества равны.
        return true;
    }
}

/*
 * SingletonSet - конкретный подкласс AbstractEnumerableSet.
 * Одноэлементное множество - это множество только для чтения
 * с единственным членом.
 */
class SingletonSet extends AbstractEnumerableSet {
    constructor(member) {
        super();
        this.member = member;
    }

    // Мы реализуем следующие три метода и унаследуем реализации
    // isEmpty, equals() и tring().
    has(x) { return x === this.member; }
    get size() { return 1; }
    *[Symbol.iterator]() { yield this.member; }
}
```

```
/*
 * AbstractWritableSet - абстрактный подкласс AbstractEnumerableSet.
 * Он определяет абстрактные методы insert() и remove(), которые вставляют
 * и удаляют индивидуальные элементы из множества, и реализует конкретные
 * методы add(), subtract() и intersect(). Обратите внимание, что здесь
 * наш API-интерфейс отличается от стандартного класса Set в JavaScript.
 */
class AbstractWritableSet extends AbstractEnumerableSet {
    insert(x) { throw new Error("Абстрактный метод"); }
    remove(x) { throw new Error("Абстрактный метод"); }

    add(set) {
        for(let element of set) {
            this.insert(element);
        }
    }

    subtract(set) {
        for(let element of set) {
            this.remove(element);
        }
    }

    intersect(set) {
        for(let element of this) {
            if (!set.has(element)) {
                this.remove(element);
            }
        }
    }
}

/**
 * BitSet - конкретный подкласс AbstractWritableSet с очень эффективной
 * реализацией множества фиксированного размера, предназначенной для
 * множеств, чьи элементы являются неотрицательными целыми числами,
 * которые меньше определенного максимального размера.
 */
class BitSet extends AbstractWritableSet {
    constructor(max) {
        super();
        this.max = max; // Максимальное целое число, которое мы можем хранить.
        this.n = 0;     // Сколько целых чисел содержит множество.
        this.numBytes = Math.floor(max / 8) + 1; //Сколько байтов нам необходимо.
        this.data = new Uint8Array(this.numBytes);  // Байты.
    }

    // Внутренний метод для проверки, является ли значение
    // допустимым членом этого множества.
    _valid(x) { return Number.isInteger(x) && x >= 0 && x <= this.max; }

    // Проверяет, установлен ли указанный бит указанного байта
    // в нашем массиве данных. Возвращает true или false.
    _has(byte, bit) { return (this.data[byte] & BitSet.bits[bit]) !== 0; }
```

```
    // Находится ли значение x в этом BitSet?
    has(x) {
        if (this._valid(x)) {
            let byte = Math.floor(x / 8);
            let bit = x % 8;
            return this._has(byte, bit);
        } else {
            return false;
        }
    }
    // Вставляет значение x в BitSet.
    insert(x) {
        if (this._valid(x)) {              // Если значение допустимо,
            let byte = Math.floor(x / 8);  // тогда преобразовать в байт и бит.
            let bit = x % 8;
            if (!this._has(byte, bit)) {   // Если этот бит еще не установлен,
                this.data[byte] |= BitSet.bits[bit];  // тогда установить его
                this.n++;                  // и инкрементировать размер множества.
            }
        } else {
            throw new TypeError("Недопустимый элемент множества: " + x );
        }
    }
    remove(x) {
        if (this._valid(x)) {              // Если значение допустимо,
            let byte = Math.floor(x / 8);  // тогда вычислить байт и бит.
            let bit = x % 8;
            if (this._has(byte, bit)) {    // Если этот бит уже установлен,
                this.data[byte] &= BitSet.masks[bit];  // тогда сбросить его
                this.n--;                  // и декрементировать размер.
            }
        } else {
            throw new TypeError("Недопустимый элемент множества: " + x );
        }
    }
    // Метод получения для возврата размера множества.
    get size() { return this.n; }
    // Выполняет итерацию по множеству, просто проверяя каждый бит по очереди
    // (Мы могли бы быть гораздо искуснее и значительно оптимизировать это.)
    *[Symbol.iterator]() {
        for(let i = 0; i <= this.max; i++) {
            if (this.has(i)) {
                yield i;
            }
        }
    }
}
// Некоторые заранее рассчитанные значения, используемые методами has(),
// insert() и remove():
BitSet.bits = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
BitSet.masks = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);
```

9.6. Резюме

В главе объяснялись ключевые возможности классов JavaScript.

- Объекты, которые являются членами одного класса, наследуют свойства от того же самого объекта прототипа. Объект прототипа является ключевой особенностью классов JavaScript, и классы можно определять только с помощью метода `Object.create()`.
- До выхода ES6 классы обычно определялись путем определения сначала функции конструктора. Функции, созданные с ключевым словом `function`, имеют свойство `prototype`, значением которого является объект, используемый в качестве прототипа всех объектов, которые создаются, когда эта функция вызывается с ключевым словом `new` как конструктор. Инициализируя упомянутый объект прототипа, вы можете определять разделяемые методы своего класса. Хотя объект прототипа — ключевая особенность класса, функция конструктора представляет собой открытую идентичность класса.
- В ES6 появилось ключевое слово `class`, которое облегчает определение классов, но "за кулисами" механизм конструкторов и прототипов остается прежним.
- Подклассы определяются с применением ключевого слова `extends` в объявлении класса.
- Подклассы могут вызывать конструкторы или переопределенные методы своих суперклассов с помощью ключевого слова `super`.

ГЛАВА 10

Модули

Цель модульного программирования — позволить собирать крупные программы с использованием модулей кода от разных авторов и источников и обеспечить корректное выполнение всего кода даже при наличии кода, которые различные авторы модулей не предвидели. На практике модульность главным образом касается инкапсуляции или сокрытия внутренних деталей реализации и поддержания порядка в глобальном пространстве имен, чтобы модули не могли случайно модифицировать переменные, функции и классы, определяемые другими модулями.

До недавнего времени в JavaScript не было никакой встроенной поддержки для модулей, и программисты, работающие над крупными кодовыми базами, из всех сил старались задействовать слабую модульность, доступную через классы, объекты и замыкания. Модульность на основе замыканий с поддержкой со стороны инструментов пакетирования кода привела к практической форме модульности, основанной на функции `require()`, которая была принята в Node. Модули на основе `require()` являются фундаментальной частью программной среды Node, но не принимались как официальная часть языка JavaScript. Взамен в ES6 модули определяются с применением ключевых слов `import` и `export`. Хотя `import` и `export` считались частью языка в течение многих лет, они лишь относительно недавно были реализованы веб-браузерами и Node. Кроме того, с практической точки зрения модульность JavaScript по-прежнему опирается на инструменты пакетирования кода.

В разделах далее в главе раскрываются следующие темы:

- самодельные модули, использующие классы, объекты и замыкания;
- модули Node, применяющие `require()`;
- модули ES6, использующие `export`, `import` и `import()`.

10.1. Модули, использующие классы, объекты и замыкания

Несмотря на возможную очевидность, стоит отметить, что одна из важных особенностей классов заключается в том, что они могут действовать как модули для своих методов. Вспомните пример 9.8, где определялся набор классов, которые все имели метод по имени `has()`. Но вы без каких-либо проблем можете написать программу, в которой применяется несколько классов множеств из данного примера: скажем, нет никакой опасности того, что реализация `has()` из `SingletonSet` заместит метод `has()` класса `BitSet`.

Причина независимости методов одного класса от методов другого несвязанного класса кроется в том, что методы каждого класса определяются как свойства независимых объектов прототипов.

Причина модульности классов объясняется модульностью объектов: определение свойства в объекте JavaScript во многом похоже на объявление переменной, но добавление свойств в объекты не влияет ни на глобальное пространство имен программы, ни на свойства других объектов. В JavaScript определено довольно много математических функций и констант, но вместо того, чтобы определить их всех глобально, они сгруппированы в виде свойств единственного глобального объекта `Math`. Ту же самую методику можно было бы использовать в примере 9.8. Вместо определения глобальных классов с именами вроде `SingletonSet` и `BitSet` можно было бы определить только один глобальных объект `Sets` со свойствами, ссылающимися на различные классы. Тогда пользователи библиотеки `Sets` могли бы ссылаться на классы с именами наподобие `Sets.Singleton` и `Sets.Bit`.

Применение классов и объектов для обеспечения модульности — распространенная и удобная методика при программировании на JavaScript, но ее недостаточно. В частности, она не предлагает нам какого-нибудь способа сокрытия внутренних деталей реализации в рамках модуля. Снова обратимся к примеру 9.8. Если бы мы писали его как модуль, то вероятно захотели бы сохранить разнообразные абстрактные классы внутренними по отношению к модулю, сделав доступными пользователям модуля только конкретные подклассы. Аналогично в классе `BitSet` методы `_valid()` и `_has()` являются внутренними утилитами, которые на самом деле не должны быть видны пользователям класса. А `BitSet.bits` и `BitSet.masks` представляют собой детали реализации, которые лучше скрыть.

Как было показано в разделе 8.6, локальные переменные и вложенные функции, определенные внутри функции, являются закрытыми в рамках этой функции. Таким образом, мы можем использовать немедленно вызываемые выражения функций для обеспечения своего рода модульности, оставляя детали реализации и служебные функции скрытыми внутри включающей функции, но делая открытый API-интерфейс модуля возвращаемым значением функции. В случае класса `BitSet` мы могли бы структурировать модуль примерно так:

```
const BitSet = (function() {  // Установить BitSet в возвращаемое
                              // значение этой функции
    // Здесь находятся закрытые детали реализации
    function isValid(set, n) { ... }
    function has(set, byte, bit) { ... }
    const BITS = new Uint8Array([1, 2, 4, 8, 16, 32, 64, 128]);
    const MASKS = new Uint8Array([~1, ~2, ~4, ~8, ~16, ~32, ~64, ~128]);

    // Открытый API-интерфейс модуля - это просто класс BitSet,
    // который мы здесь определяем и возвращаем.
    // Класс может использовать закрытые функции и константы,
    // определенные выше, но они будут скрыты от пользователей класса.
    return class BitSet extends AbstractWritableSet {
        // ... реализация не показана ...
    };
}());
```

Такой подход к модульности становится чуть более интересным, когда в модуле содержится несколько элементов. Например, в следующем коде определяется мини-модуль расчета статистических данных, который экспортирует функции `mean()` и `stddev()`, оставляя детали реализации скрытыми:

```
// Вот так мы могли бы определить модуль расчета статистических данных.
const stats = (function() {
    // Служебные функции закрыты по отношению к модулю.
    const sum = (x, y) => x + y;
    const square = x => x * x;
    // Открытая функция, которая будет экспортироваться.
    function mean(data) {
        return data.reduce(sum)/data.length;
    }
    // Открытая функция, которая будет экспортироваться.
    function stddev(data) {
        let m = mean(data);
        return Math.sqrt(
            data.map(x => x - m).map(square).reduce(sum)/(data.length-1)
        );
    }
    // Открытые функции экспортируются в виде свойств объекта.
    return { mean, stddev };
}());
// А так мы можем использовать модуль.
stats.mean([1, 3, 5, 7, 9])     // => 5
stats.stddev([1, 3, 5, 7, 9])   // => Math.sqrt(10)
```

10.1.1. Автоматизация модульности на основе замыканий

Обратите внимание, что трансформация файла кода JavaScript в модуль такого вида за счет вставки определенного текста в начале и конце файла, является явно механическим процессом. Все, что необходимо — это определенное соглашение для файла кода JavaScript, которое бы указывало, какие значения должны экспортироваться, а какие нет.

Представим себе инструмент, который берет набор файлов, помещает содержимое каждого файла внутрь немедленно вызываемого выражения функции, отслеживает возвращаемое выражение каждой функции и объединяет все в один большой файл. Результат может выглядеть следующим образом:

```
const modules = {};
function require(moduleName) { return modules[moduleName]; }
modules["sets.js"] = (function() {
    const exports = {};
    // Здесь находится содержимое файла sets.js:
    exports.BitSet = class BitSet { ... };

    return exports;
}());
modules["stats.js"] = (function() {
    const exports = {};

    // Здесь находится содержимое файла stats.js:
    const sum = (x, y) => x + y;
    const square = x = > x * x;
    exports.mean = function(data) { ... };
    exports.stddev = function(data) { ... };
    return exports;
}());
```

Имея модули, пакетированные в единственном файле вроде показанного в предыдущем примере, вы можете также представить, что для использования модулей подойдет код такого вида:

```
// Получить ссылки на необходимые модули (или на содержимое модуля):
const stats = require("stats.js");
const BitSet = require("sets.js").BitSet;

// Код для использования этих модулей:
let s = new BitSet(100);
s.insert(10);
s.insert(20);
s.insert(30);
let average = stats.mean([...s]);  // средняя величина равна 20
```

Код является грубым наброском того, как работают инструменты пакетирования кода (наподобие webpack и Parcel) для веб-браузеров, а также простым введением в функцию `require()` вроде той, что применяется в программах Node.

10.2. Модули в Node

При программировании с использованием Node вполне нормально разбивать программы на столько файлов, сколько кажется естественным. Предполагается, что такие файлы кода JavaScript хранятся в быстрой файловой системе. В отличие от веб-браузеров, которые обязаны читать файлы кода JavaScript по относительно медленному сетевому подключению, отсутствует необходимость или преимущество от пакетирования программы Node в единственный файл JavaScript.

Каждый файл в Node является независимым модулем с закрытым пространством имен. Константы, переменные, функции и классы, определенные в одном файле, будут закрытыми в рамках этого файла, если только не экспортировать их. Значения, экспортированные одним модулем, будут видны в другом модуле, только если другой модуль явно импортирует их.

Модули Node импортируют другие модули с помощью функции `require()` и экспортируют свои открытые API-интерфейсы, устанавливая свойства объекта `exports` или полностью замещая объект `module.exports`.

10.2.1. Экспортирование в Node

В Node определен глобальный объект `exports`, который доступен всегда. Если вы пишете модуль Node, который экспортирует множество значений, тогда можете просто присвоить их свойствам объект `exports`:

```
const sum = (x, y) => x + y;
const square = x => x * x;
exports.mean = data => data.reduce(sum)/data.length;
exports.stddev = function(d) {
  let m = exports.mean(d);
  return
    Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1));
};
```

Однако часто вы хотите определить модуль, который экспортирует только одну функцию или класс, а не объект, полный функций или классов. Для этого вы присваиваете одиночное значение, которое желаете экспортировать, объекту `module.exports`:

```
module.exports = class BitSet extends AbstractWritableSet {
   // реализация не показана
};
```

Стандартным значением `module.exports` является тот же объект, на который ссылается `exports`. В предыдущем модуле расчета статистических данных (`stats`) мы могли бы присвоить функцию вычисления средней величины `module.exports.mean`, а не `exports.mean`. Еще один подход работы с модулями наподобие `stats` предусматривает экспортирование одиночного объекта в конце модуля вместо экспортирования функций друг за другом:

```
// Определить все функции, открытые и закрытые.
const sum = (x, y) => x + y;
const square = x => x * x;
const mean = data => data.reduce(sum)/data.length;
const stddev = d => {
  let m = mean(d);
  return
    Math.sqrt(d.map(x => x - m).map(square).reduce(sum)/(d.length-1));
};

// Теперь экспортировать только открытые функции.
module.exports = { mean, stddev };
```

10.2.2. Импортирование в Node

Модуль Node импортирует другой модуль, вызывая функцию `require()`. Аргументом указанной функции будет имя модуля, подлежащего импортированию, а возвращаемым значением — любое значение (обычно функция, класс или объект), которое модуль экспортирует.

Если вы хотите импортировать системный модуль, встроенный в Node, или модуль, установленный в вашей системе через диспетчер пакетов, тогда просто применяете неуточненное имя модуля, без символов /, которые превратили бы его в путь внутри файловой системы:

```
// Эти модули встроены в Node.
const fs = require("fs");       // Встроенный модуль файловой системы
const http = require("http");   // Встроенный модуль HTTP
// Фреймворк HTTP-сервера Express - сторонний модуль.
// Не является частью Node, но был установлен локально.
const express = require("express");
```

Когда вы желаете импортировать модуль собственного кода, то имя модуля должно быть путем к файлу, содержащему ваш код, относительно файла текущего модуля. Допускается использовать абсолютные пути, которые начинаются с символа /, но обычно при импортировании модулей, являющихся частью вашей программы, имена модулей будут начинаться с `./` либо временами с `../`, указывая на то, что они относительны текущего или родительского каталога. Например:

```
const stats = require('./stats.js');
const BitSet = require('./utils/bitset.js');
```

(Вы также можете опускать суффикс `.js` в импортируемых файлах и Node по-прежнему отыщет файлы, но эти файловые расширения часто включаются явно.)

Когда модуль экспортирует лишь одну функцию или класс, то вам понадобится только затребовать его посредством `require()`. Если модуль экспортирует объект с множеством свойств, тогда у вас есть выбор: вы можете импортировать целый объект или просто специфические свойства (применяя деструктурирующее присваивание) объекта, которые планируете использовать. Сравните два описанных подхода:

```
// Импортировать целый объект stats со всеми его функциями.
const stats = require('./stats.js');

// У нас больше функций, чем нужно, но они аккуратно
// организованы в удобное пространство имен stats.
let average = stats.mean(data);

// В качестве альтернативы мы можем использовать идиоматическое
// деструктурирующее присваивание для импортирования именно тех функций,
// которые мы хотим поместить прямо в локально пространство имен:
const { stddev } = require('./stats.js');

// Это элегантно и лаконично, хотя без префикса stats мы теряем
// немного контекста как пространства имен для функции stddev().
let sd = stddev(data);
```

10.2.3. Модули в стиле Node для веб-сети

Модули с объектом `exports` и функцией `require()` встроены в Node. Но если вы хотите обрабатывать свой код инструментом пакетирования вроде webpack, тогда можете применять такой стиль модулей в коде, который предназначен для запуска в веб-браузерах. До недавнего времени это было обычным делом, и вы можете встретить много кода, ориентированного на веб-сеть, где по-прежнему так поступают.

Тем не менее, в настоящее время, когда в JavaScript имеется собственный стандартный синтаксис модулей, разработчики, использующие инструменты пакетирования, с большей вероятностью будут работать с официальными модулями JavaScript, применяя операторы `import` и `export`.

10.3. Модули в ES6

Стандарт ES6 добавил в JavaScript ключевые слова `import` и `export` и окончательно обеспечил поддержку подлинной модульности как основного языкового средства. Модульность ES6 концептуально такая же, как модульность Node: каждый файл является отдельным модулем, а определенные внутри файла константы, переменные, функции и классы закрыты по отношению к этому модулю, если только явно не экспортируются. Значения, экспортированные из одного модуля, будут доступны для использования в модулях, которые явно их импортируют. Модули ES6 отличаются от модулей Node синтаксисом, применяемым для экспортирования и импортирования, а также способом определения модулей в веб-браузерах. Все аспекты подробно обсуждаются в последующих подразделах.

Однако первым делом обратите внимание, что модули ES6 также отличаются от обычных "сценариев" JavaScript в некоторых важных отношениях. Наиболее очевидной разницей является сама модульность: в обычных сценариях объявления на верхнем уровне переменных, функций и классов помещаются в единственный глобальный контекст, разделяемый всеми сценариями. В случае модулей каждый файл имеет собственный закрытый контекст и может использовать операторы `import` и `export`, в чем по большому счету и заключается весь смысл. Но между модулями и сценариями существуют и другие отличия. Код внутри модуля ES6 (подобно коду внутри определения `class` в ES6) автоматически находится в строгом режиме (см. подраздел 5.6.3). Это означает, что когда вы начнете применять модули ES6, то вам больше никогда не придется записывать `"use strict"`.

Кроме того, в коде модулей нельзя использовать оператор `with`, объект `arguments` или необъявленные переменные. Модули ES6 даже немного строже строгого режима: в строгом режиме в функциях, вызываемых как функции, `this` равно `undefined`. В модулях `this` равно `undefined` даже в коде верхнего уровня. (Напротив, сценарии в веб-браузерах и Node устанавливают `this` в глобальный объект.)

Модули ES6 в веб-сети и в Node

Модули ES6 применялись в веб-сети в течение многих лет с помощью инструментов пакетирования кода вроде webpack, которые объединяют независимые модули кода JavaScript в крупные немодульные пакеты, подходящие для включения внутрь веб-страниц. Тем не менее, на момент написания книги модули ES6 в конечном итоге стали естественным образом поддерживаться всеми веб-браузерами кроме Internet Explorer. В случае естественного использования модули ES6 добавляются к HTML-страницам посредством специального дескриптора `<script type="module">`, который будет описан позже в главе.

Между тем, впервые применив модульность JavaScript, среда Node оказалась в неловком положении, когда ей пришлось поддерживать две не полностью совместимые системы модулей. В версии Node 13 поддерживаются модули ES6, но на данный момент подавляющее большинство программ Node по-прежнему используют модули Node.

10.3.1. Экспортирование в ES6

Чтобы экспортировать константу, переменную, функцию или класс из модуля ES6, просто добавьте перед объявлением ключевое слово `export`:

```
export const PI = Math.PI;

export function degreesToRadians(d) { return d * PI / 180; }

export class Circle {
   constructor(r) { this.r = r; }
   area() { return PI * this.r * this.r; }
}
```

Вместо разбрасывания ключевых слов `export` по всему модулю вы можете определить свои константы, переменные, функции и классы нормальным образом, не указывая `export`, после чего (обычно в конце модуля) записать единственный оператор `export`, который описывает экспортируемые значения в одном месте. Таким образом, взамен трех отдельных ключевых слов `export` в предыдущем коде можно написать одну эквивалентную строку в конце:

```
export { Circle, degreesToRadians, PI };
```

Синтаксис выглядит как ключевое слово `export`, за которым следует объектный литерал (в сокращенной записи). Но в данном случае фигурные скобки на самом деле не определяют объектный литерал. Синтаксис экспортирования просто требует списка разделенных запятыми идентификаторов в фигурных скобках.

Часто пишут модули, которые экспортируют только одно значение (как правило, функцию или класс), и в таком случае обычно применяется `export default`, а не `export`:

```
export default class BitSet {
   // реализация не показана
}
```

Экспорт по умолчанию чуть легче импортировать, чем экспорт не по умолчанию, и потому при наличии только одного экспортируемого значения использование `export default` облегчает работу для модулей, которые задействуют ваше экспортируемое значение.

Обычный экспорт посредством `export` может делаться только в объявлениях, которые имеют имя. Экспорт по умолчанию с помощью `export default` способен экспортировать любое выражение, включая выражения анонимных функций и выражения анонимных классов. Это означает, что если вы применяете `export default`, то можете экспортировать объектные литералы. Таким образом, в отличие от синтаксиса `export`, когда вы видите фигурные скобки после `export default`, они действительно представляют собой литерал, который экспортируется.

Для модулей вполне законно, хотя слегка необычно, иметь набор обыкновенных экспортируемых значений и также экспорт по умолчанию. Если в модуле есть экспорт по умолчанию, тогда он может быть только один.

Наконец, обратите внимание, что ключевое слово `export` может находиться только на верхнем уровне кода JavaScript. Экспортировать значение изнутри класса, функции, цикла или условного оператора нельзя. (Это важная особенность системы модулей ES6, которая позволяет проводить статический анализ: экспорт модулей будет одинаковым при каждом запуске, а экспортируемые символы могут определяться перед фактическим выполнением модуля.)

10.3.2. Импортирование в ES6

Значения, которые были экспортированы другими модулями, импортируются с помощью ключевого слова `import`. Простейшая форма импортирования используется для модулей, определяющих экспорт по умолчанию:

```
import BitSet from './bitset.js';
```

Здесь указывается ключевое слово `import`, за ним следует идентификатор, далее ключевое слово `from` и в конце строковый литерал, задающий имя модуля, из которого импортируется экспорт по умолчанию. Значение экспорта по умолчанию указанного модуля становится значением указанного идентификатора в текущем модуле.

Идентификатор, которому присваивается импортированное значение, является константой, как если бы он был объявлен с ключевым словом `const`. Подобно экспорту импорт может находиться только на верхнем уровне модуля, и он не разрешен внутри классов, функций, циклов или условных операторов. По почти универсальному соглашению операторы импорта, необходимые модулю, размещаются в начале модуля. Однако интересно отметить, что поступать так вовсе не обязательно: подобно объявлениям функций операторы импорта "поднимаются" в начало и все импортированные значения доступны для любого выполняемого кода модуля.

Модуль, из которого импортируется значение, указывается как константный строковый литерал в одинарных или двойных кавычках. (Вы не можете применять переменную или другое выражение, чьим значением является строка, и не

можете использовать строку внутри обратных кавычек, т.к. шаблонные литералы способны вставлять переменные и не всегда иметь константные значения.) В веб-браузерах эта строка интерпретируется как URL относительно местоположения модуля, который делает импортирование. (В среде Node или в случае применения инструмента пакетирования строка интерпретируется как имя файла относительно текущего модуля, но на практике разница едва заметна.) Строка *спецификатора модуля* должна быть абсолютным путем, начинающимся с `/`, относительным путем, начинающимся с `./` либо `../`, или полным URL с протоколом и именем хоста. Стандарт ES6 не разрешает использовать неуточненную строку спецификатора модуля, такую как `"util.js"`, поскольку неясно, что она задает — имя модуля, находящегося в том же самом каталоге, где и текущий модуль, либо имя системного модуля, который установлен в каком-то особом месте. (Такое ограничение против "голых спецификаторов модулей" не соблюдается инструментами пакетирования кода вроде webpack, которые можно легко сконфигурировать на поиск модулей в указанном библиотечном каталоге.) В будущей версии языка могут быть разрешены "голые спецификаторы модулей", но в данный момент они не допускаются. Если вы хотите импортировать модуль из каталога, в котором находится текущий модуль, тогда просто поместите `./` перед именем модуля и импортируйте из `./util.js`, а не `util.js`.

До сих пор мы рассматривали случай импортирования одиночного значения из модуля, в котором применяется `export default`. Чтобы импортировать значения из модуля, экспортирующего множество значений, мы используем слегка отличающийся синтаксис:

```
import { mean, stddev } from "./stats.js";
```

Вспомните, что экспорт по умолчанию не обязательно должен иметь имя в модуле, в котором он определен. Взамен при импортировании значений мы предоставляем локальные имена. Но экспорт не по умолчанию модуля имеет имя в экспортирующем модуле, и при импортировании таких значений мы ссылаемся на них по этим именам. Экспортирующий модуль может экспортировать любое количество именованных значений. Оператор `import`, который ссылается на данный модуль, может импортировать любое подмножество значений, перечисляя их имена внутри фигурных скобок. Фигурные скобки делают оператор `import` подобного рода выглядящим как деструктурирующее присваивание, которое действительно является хорошей аналогией того, что выполняет такой стиль импортирования. Все идентификаторы внутри фигурных скобок поднимаются в начало импортирующего модуля и ведут себя как константы.

Руководства по стилю иногда рекомендуют явно импортировать каждый символ, который будет применяться в текущем модуле. Тем не менее, при импортировании из модуля, где определено много экспортированных значений, можно легко импортировать их всех посредством следующего оператора `import`:

```
import * as stats from "./stats.js";
```

Оператор `import` такого вида создает объект и присваивает его константе по имени `stats`. Каждый экспорт не по умолчанию модуля становится свойством объекта `stats`. Экспорт не по умолчанию всегда имеет имя, которое использу-

ется в качестве имени свойства внутри объекта. Свойства на самом деле будут константами: их нельзя переопределять или удалять. Благодаря показанному в предыдущем примере групповому импортированию импортирующий модуль будет работать с импортированными функциями `mean()` и `stddev()` через объект `stats`, вызывая их как `stats.mean()` и `stats.stddev()`.

По обыкновению модули определяют либо один экспорт по умолчанию, либо множество именованных экспортов. Для модуля вполне законно, хотя и несколько необычно, применять и `export`, и `export default`. Но когда подобное происходит, то вы можете импортировать значение по умолчанию и именованные значения с помощью оператора `import` такого вида:

```
import Histogram, { mean, stddev } from "./histogram-stats.js";
```

До сих пор вы видели, как импортировать из модулей с экспортом по умолчанию и из модулей с экспортом не по умолчанию либо с именованным экспортом. Но есть еще одна форма оператора `import`, которая используется с модулями, вообще не имеющими операторов экспорта. Чтобы включить модуль без операторов экспорта в свою программу, необходимо просто указать ключевое слово `import` со спецификатором модуля:

```
import "./analytics.js";
```

Такой модуль запускается при первом его импортировании. (Последующее импортирование ничего не делает.) Модуль, который всего лишь определяет функции, полезен только в случае экспортирования, по меньшей мере, одной из них. Но если модуль выполняет какой-то код, тогда он может быть полезным даже без символов. Модуль аналитики для веб-приложения мог бы выполнять код для регистрации разнообразных обработчиков событий и затем применять эти обработчики событий с целью отправки серверу телеметрических данных в назначенные моменты времени. Модуль аналитики является самодостаточным и не обязан что-то экспортировать, но нам все равно его необходимо импортировать, чтобы он фактически выполнялся как часть программы.

Обратите внимание, что вы можете использовать этот ничего не импортирующий синтаксис `import` даже с модулями, которые имеют экспорт. Если модуль определяет полезное поведение, независимое от экспортируемых значений, а ваша программа не нуждается в таких значениях, то вы по-прежнему можете импортировать модуль просто ради такого поведения по умолчанию.

10.3.3. Импортирование и экспортирование с переименованием

Если два модуля экспортируют два разных значения с указанием одного и того же имени, но вы желаете импортировать оба значения, тогда при импортировании вам придется переименовать одно или оба значения. Аналогично, если вы хотите импортировать значение, имя которого уже задействовано в вашем модуле, то должны будете переименовать импортированное значение. Вы можете применять ключевое слово `as` с именованным импортом, чтобы переименовать значение при его импортировании:

```
import { render as renderImage } from "./imageutils.js";
import { render as renderUI } from "./ui.js";
```

Приведенные строки кода импортируют две функции в текущий модуль. Обе функции имеют имя `render()` в модулях, где они определены, но импортируются с более описательными и устраняющими неоднозначность именами `renderImage()` и `renderUI()`.

Вспомните, что экспорт по умолчанию не имеет имени. При импортировании экспорта по умолчанию импортирующий модуль всегда выбирает имя. Таким образом, в этом случае нет никакой необходимости в специальном синтаксисе.

Однако следует отметить, что возможность переименования при импорте предоставляет еще один способ импортирования из модулей, которые определяют экспорт по умолчанию и именованные экспорты. Возьмем модуль `"./histogram-stats.js"` из предыдущего раздела. Вот как по-другому импортировать экспорт по умолчанию и именованные экспорты данного модуля:

```
import { default as Histogram, mean, stddev } from "./histogram-stats.js";
```

В таком случае ключевое слово `default` языка JavaScript служит заполнителем и позволяет указать, что мы хотим импортировать и назначить имя экспорту по умолчанию модуля.

Также можно переименовывать значения при их экспорте, но только когда используется вариант оператора `export` с фигурными скобками. В этом обычно нет необходимости, но если вы выбрали короткие имена для применения внутри своего модуля, то можете предпочесть экспортировать нужные значения с более описательными именами, которые с меньшей вероятностью будут конфликтовать с именами в других модулях. Как и при импорте для такой цели используется ключевое слово `as`:

```
export {
    layout as calculateLayout,
    render as renderLayout
};
```

Имейте в виду, что хотя фигурные скобки выглядят похожими на объектные литералы, они таковыми не являются, а ключевое слово `export` ожидает перед `as` одиночный идентификатор, но не выражение. К сожалению, сказанное означает, что вы не можете применять переименование экспорта, подобное приведенному ниже:

```
export { Math.sin as sin, Math.cos as cos };  // Синтаксическая ошибка
```

10.3.4. Повторное экспортирование

В настоящей главе мы обсуждали гипотетический модуль `"./stats.js"`, который экспортирует функции `mean()` и `stddev()`. Если бы мы писали такой модуль и полагали, что многим его пользователям понадобится одна функция или другая, тогда мы могли бы определить `mean()` в модуле `"./stats/mean.js"` и `stddev()` в модуле `"./stats/stddev.js"`. Таким образом, программам необходимо импортировать только нужные функции и не раздувать свой код, импортируя то, что не требуется.

Тем не менее, даже если мы определили эти статистические функции в отдельных модулях, то могли бы ожидать, что будет предостаточно программ, которых интересуют обе функции и которые оценят удобный модуль `"./stats.js"`, откуда можно импортировать две функции в одной строке кода.

С учетом того, что реализации теперь находятся в разных файлах, определить файл `"./stat.js"` просто:

```
import { mean } from "./stats/mean.js";
import { stddev } from "./stats/stddev.js";
export { mean, stdev };
```

Модули ES6 предугадывают такой сценарий использования и предлагают для него специальный синтаксис. Вместо импортирования символа всего лишь для того, чтобы экспортировать его вновь, вы можете объединить шаги импорта и экспорта в единственный оператор "повторного экспорта", в котором применяются ключевые слова `export` и `from`:

```
export { mean } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

Обратите внимание, что имена `mean` и `stddev` на самом деле в коде не используются. Если нас не интересует избирательность при повторном экспортировании, и мы просто хотим экспортировать из другого модуля все именованные значения, тогда можем применить групповой экспорт:

```
export * from "./stats/mean.js";
export * from "./stats/stddev.js";
```

Синтаксис повторного экспортирования позволяет переименовывать с помощью `as` в точности как обыкновенные операторы `import` и `export`. Допустим, нам нужно повторно экспортировать функцию `mean()`, но также определить для нее еще одно имя `average()`. Вот что можно сделать:

```
export { mean, mean as average } from "./stats/mean.js";
export { stddev } from "./stats/stddev.js";
```

Во всех повторных экспортах в этом примере предполагается, что модули `"./stats/mean.js"` и `"./stats/stddev.js"` экспортируют свои функции с использованием `export`, а не `export default`. Однако фактически из-за того, что они являются модулями только с одним экспортом, имело бы смысл определить их посредством `export default`. Если мы поступим так, тогда синтаксис повторного экспорта станет чуть сложнее, поскольку необходимо определить имя для неименованных экспортов по умолчанию. Мы можем достичь цели примерно так:

```
export { default as mean } from "./stats/mean.js";
export { default as stddev } from "./stats/stddev.js";
```

Если вы хотите повторно экспортировать именованный символ из другого модуля как экспорт по умолчанию своего модуля, то можете применить ключевое слово `import`, за которым следует `export default`, или объединить два оператора, как показано ниже:

```
// Импортировать функцию mean() из ./stats.js и сделать
// ее экспортом по умолчанию этого модуля.
export { mean as default } from "./stats.js"
```

Наконец, чтобы повторно экспортировать экспорт по умолчанию другого модуля как экспорт по умолчанию своего модуля (хотя неясно, для чего это может понадобиться, ведь пользователи в состоянии импортировать другой модуль напрямую), вы можете написать:

```
// Модуль average.js просто повторно экспортирует экспорт
// по умолчанию stats/mean.js.
export { default } from "./stats/mean.js"
```

10.3.5. Модули JavaScript для веб-сети

В предшествующих подразделах модули ES6 вместе со своими объявлениями `import` и `export` были описаны в несколько абстрактной манере. В текущем и следующем подразделах мы обсудим, как они на самом деле работают в веб-браузерах, и если вы не являетесь опытным разработчиком веб-приложений, то может выясниться, что оставшийся материал настоящей главы будет легче понять только после чтения главы 15.

По состоянию на начало 2020 года производственный код, использующий модули ES6, все еще обычно пакетировался с помощью инструментов, подобных webpack. С таким решением связаны компромиссы[1], но в целом пакетирование кода имеет тенденцию обеспечивать более высокую производительность. В будущем ситуация вполне может измениться, т.к. скорость сети растет, а поставщики браузеров продолжают оптимизировать своих реализации модулей ES6.

Несмотря на то что инструменты пакетирования по-прежнему могут быть желательны в производственной среде, они больше не требуются при разработке, поскольку текущие версии всех браузеров предлагают встроенную поддержку для модулей JavaScript. Вспомните, что по умолчанию модули применяют строгий режим, `this` не ссылается на глобальный объект и объявления верхнего уровня не разделяются глобально. Из-за того, что модули должны выполняться иначе, чем унаследованный немодульный код, их введение требует внесения изменений в код HTML и JavaScript. Если вы хотите естественным образом использовать директивы `import` в веб-браузере, тогда обязаны сообщить веб-браузеру о том, что ваш код является модулем, с применением дескриптора `<script type="module">`.

Одна из элегантных особенностей модулей ES6 заключается в том, что каждый модуль имеет статический набор импортов. Таким образом, имея единственный стартовый модуль, веб-браузер может загрузить все импортированные модули, затем все модули, импортированные первым пакетом модулей, и так далее до тех пор, пока не будет загружена полная программа. Вы видели, что

[1] Например: веб-приложения, которые подвергаются частым инкрементным обновлениям, и пользователи, часто обращающиеся к ним снова, могут обнаружить, что применение небольших модулей вместо крупных пакетов может привести к улучшению среднего времени загрузки из-за лучшего использования кеша пользовательского браузера.

спецификатор модуля в операторе import может трактоваться как относительный URL. Дескриптор <script type="module"> отмечает начальную точку модульной программы. Тем не менее, импортируемые им модули не должны находиться в дескрипторах <script>: взамен они загружаются по запросу как обыкновенные файлы JavaScript и выполняются в строгом режиме как обычные модули ES6. Использовать дескриптор <script type="module"> для определения главной точки входа в модульной программе JavaScript довольно просто, например:

```
<script type="module">import "./main.js";</script>
```

Код внутри встроенного дескриптора <script type="module"> представляет собой модуль ES6 и как таковой может применять оператор export. Однако поступать так не имеет смысла, потому что синтаксис HTML-дескриптора <script> не предоставляет какого-либо способа определения имен для встроенных модулей, поэтому даже если такой модуль экспортирует значение, то у другого модуля нет никакой возможности его импортировать.

Сценарии с атрибутом type="module" загружаются и выполняются подобно сценариям с атрибутом defer. Загрузка кода начинается, как только синтаксический анализатор HTML встречает дескриптор <script> (в случае модулей такой шаг загрузки кода может быть рекурсивным процессом, загружающим множество файлов JavaScript). Но выполнение кода не начнется до тех пор, пока не закончится синтаксический анализ HTML-разметки. После того, как синтаксический анализ HTML-разметки завершен, сценарии (модульные и немодульные) выполняются в порядке, в котором они появлялись в HTML-документе.

Вы можете изменить время выполнения модулей посредством атрибута async, который делает для модулей то же самое, что и для обычных сценариев. Модуль async будет выполняться, как только загрузится код, даже если синтаксический анализ HTML-разметки не закончен и даже если это меняет относительный порядок следования сценариев.

Веб-браузеры, поддерживающие <script type="module">, также должны поддерживать <script nomodule>. Браузеры, не осведомленные о модулях, игнорируют любой сценарий с атрибутом nomodule и выполнять его не будут. Браузеры, которые не поддерживают модули, не распознают атрибут nomodule, а потому проигнорируют его и запустят сценарий. В итоге мы получаем мощный способ решения проблем с совместимостью браузеров. Браузеры, поддерживающие модули ES6, также поддерживают другие современные возможности JavaScript вроде классов, стрелочных функций и цикла for/of. Если вы пишете современный код JavaScript и загружаете его с помощью <script type="module">, то знаете, что он будет загружаться только браузерами, которые способны его поддерживать. И в качестве альтернативного варианта для IE11 (который 2020 году фактически остается единственным браузером, не поддерживающим ES6) вы можете использовать инструменты наподобие Babel и webpack, чтобы преобразовать свой код в немодульный код ES5, и затем загрузить менее эффективный трансформированный код через <script nomodule>.

Еще одно важное отличие между обыкновенными сценариями и модульными сценариями связано с загрузкой из разных источников. Обычный дескриптор `<script>` будет загружать файл кода JavaScript из любого сервера в Интернете и от данного факта зависит инфраструктура Интернета, касающаяся рекламы, аналитики и кода отслеживания. Но дескриптор `<script type="module">` дает возможность ужесточить это — модули могут загружаться только из того же источника, что и вмещающий HTML-документ, или когда предусмотрены надлежащие заголовки CORS (cross-origin resource sharing — совместное использование ресурсов между разными источниками) для безопасного разрешения загрузки из разных источников. Прискорбный побочный эффект такого нового ограничения безопасности заключается в том, что оно затрудняет тестирование модулей ES6 в режиме разработки с применением URL вида `file:`. При использовании модулей ES6 вам, по всей видимости, придется настроить статический веб-сервер для тестирования.

Некоторым программистам нравится применять файловое расширение `.mjs`, чтобы различать свои модульные файлы JavaScript от обыкновенных немодульных файлов JavaScript с традиционным расширением `.js`. Для веб-браузеров и дескрипторов `<script>` файловое расширение в действительности не имеет значения. (Тем не менее, тип MIME важен, так что если вы используете файлы `.mjs`, тогда может возникнуть необходимость конфигурирования вашего веб-сервера с целью их обслуживания с таким же типом MIME, как у файлов `.js`.) Поддержка ES6 в среде Node применяет файловые расширения как подсказку для распознавания, какая система модулей используется каждым загружаемым ею файлом. Таким образом, если вы пишете модули ES6 и хотите, чтобы они были пригодными к применению с Node, тогда вероятно полезно принять соглашение об именовании `.mjs`.

10.3.6. Динамическое импортирование с помощью `import()`

Вы видели, что директивы `import` и `export` в ES6 являются полностью статическими и позволяют интерпретаторам и другим инструментам JavaScript определять отношения между модулями посредством простого текстового анализа, пока модули загружаются, без фактического выполнения любого кода в модулях. С помощью статически импортируемых модулей вы гарантируете, что значения, импортируемые в модуль, будут готовы к использованию до того, как начнет выполняться любой код в вашем модуле.

В Интернете код должен передаваться по сети, а не читаться из файловой системы. После передачи код часто выполняется на мобильных устройствах с относительно медленными процессорами. Это не та среда, где статическое импортирование модулей, которое требует загрузки всей программы до того, как она запустится, имеет большой смысл.

Веб-приложения обычно первоначально загружают только такой объем кода, которого достаточно для визуализации первой страницы, отображаемой пользователю. Затем, как только пользователь располагает вводным содержимым для взаимодействия, может начинаться загрузка гораздо большего объема кода, необходимого для остатка веб-приложения. Веб-браузеры облегчают динами-

ческую загрузку кода за счет применения API-интерфейса DOM для внедрения нового дескриптора <script> в текущий HTML-документ и веб-приложения делают это в течение многих лет.

Хотя динамическая загрузка стала возможной уже давно, она не была частью самого языка. Ситуация изменилась с появлением `import()` в ES2020 (по состоянию на начало 2020 года динамическая загрузка поддерживалась всеми браузерами, которые поддерживали модули ES6). Вы передаете вызову `import()` спецификатор модуля, а он возвращает объект `Promise`, представляющий асинхронный процесс загрузки и выполнения указанного модуля. Когда динамическое импортирование завершено, объект `Promise` становится "удовлетворенным" (асинхронное программирование и объекты `Promise` рассматриваются в главе 13) и выпускает объект, подобный тому, который вы получили бы с помощью формы `import * as` оператора статического импорта.

Итак, вместо импортирования модуля `"./stats.js"` статическим образом:

```
import * as stats from "./stats.js";
```

мы могли бы импортировать его и работать с ним динамически:

```
import("./stats.js").then(stats => {
   let average = stats.mean(data);
})
```

Или же в асинхронной функции (чтобы понять приведенный ниже код, возможно, придется почитать главу 13) мы можем упростить код посредством `await`:

```
async analyzeData(data) {
   let stats = await import("./stats.js");
   return {
      average: stats.mean(data),
      stddev: stats.stddev(data)
   };
}
```

Аргументом `import()` должен быть спецификатор модуля, точно такой же, как тот, что вы использовали бы со статической директивой `import`. Но в случае применения `import()` вы не ограничены использованием константного строкового литерала: подойдет любое выражение, результатом вычисления которого будет строка в надлежащей форме.

Динамический вызов `import()` выглядит подобно вызову функции, но на самом деле это не так. Взамен `import()` является операцией, а круглые скобки представляют собой обязательную часть синтаксиса операции. Причина выбора такого слегка необычного синтаксиса связана с тем, что операция `import()` должна быть способна распознавать спецификаторы модулей как URL, указанные относительно выполняемого в текущий момент модуля, что требует определенной магии при реализации, которую нельзя было бы законно поместить в функцию JavaScript. Различие между функциями и операциями редко имеет значение на практике, но вы его заметите, если попытаетесь написать код вроде `console.log(import);` или `let require = import;`.

В заключение следует отметить, что динамическая операция `import()` предназначена не только для веб-браузеров. Она также может принести пользу

инструментам пакетирования кода наподобие webpack. Самый простой способ применения инструмента пакетирования кода — сообщить ему главную точку входа для программы и разрешить найти все статические директивы `import` и собрать все в один крупный файл. Однако, стратегически используя динамические вызовы `import()`, вы можете разбить такой монолитный пакет на набор пакетов меньшего размера, которые можно загружать по запросу.

10.3.7. `import.meta.url`

Осталось обсудить последнюю особенность системы модулей ES6. Внутри модуля ES6 (но не в обычном `<script>` или модуле Node, загруженном с помощью `require()`) специальный синтаксис `import.meta` ссылается на объект, который содержит метаданные о выполняющемся в текущий момент модуле. Свойство `url` этого объекта хранит URL, из которого модуль был загружен. (В Node это будет URL вида `file://`.)

Основной вариант применения `import.meta.url` — возможность ссылки на изображения, файлы данных и другие ресурсы, которые хранятся в том же самом каталоге, что и модуль (или в каталоге относительно него). Конструктор `URL()` облегчает распознавание URL, указанного относительно абсолютного URL наподобие `import.meta.url`. Предположим, например, что вы написали модуль, включающий строки, которые необходимо локализовать, а файлы локализации хранятся в подкаталоге `l10n/`, находящемся в том же каталоге, что и сам модуль. Модуль мог бы загружать свои строки с использованием URL, созданного с помощью функции вроде показанной ниже:

```
function localStringsURL(locale) {
    return new URL(`l10n/${locale}.json`, import.meta.url);
}
```

10.4. Резюме

Цель модульности — позволить программистам скрывать детали реализации своего кода, чтобы порции кода из различных источников можно было собирать в крупные программы, не беспокоясь о том, что одна порция перезапишет функции или переменные другой порции. В главе объяснялись три системы модулей JavaScript.

- В самом начале существования JavaScript модульности можно было достичь только за счет умелого использования немедленно вызываемых выражений функций.
- Среда Node добавляет собственную систему модулей поверх языка JavaScript. Модули Node импортируются с помощью `require()` и определяют свои экспорты, устанавливая свойства объекта `exports` или `module.exports`.
- В версии ES6 язык JavaScript, в конце концов, получил собственную систему модулей с ключевыми словами `import` и `export`, а в ES2020 добавилась поддержка динамического импортирования с помощью `import()`.

ГЛАВА 11

Стандартная библиотека JavaScript

Некоторые типы данных, такие как числа и строки (см. главу 3), объекты (см. главу 6) и массивы (см. главу 7), настолько фундаментальны в JavaScript, что мы можем считать их частью самого языка. В этой главе раскрываются другие важные, но менее фундаментальные API-интерфейсы, о которых можно думать как об определении "стандартной библиотеки" для JavaScript: они являются полезными классами и функциями, встроенными в JavaScript и доступными всем программам JavaScript в веб-браузерах и Node[1].

Разделы главы независимы друг от друга, поэтому вы можете читать их в любом порядке. Ниже перечислены охватываемые темы.

- Классы `Set` и `Map` для представления множеств значений и отображений одного множества значений на другое.
- Объекты, похожие на массивы, которые известны как типизированные массивы и представляют двоичные данные, наряду со связанным классом для извлечения значений из двоичных данных, отличающихся от массивов.
- Регулярные выражения и класс `RegExp`, которые определяют текстовые шаблоны и удобны для обработки текста. Кроме того, подробно рассматривается синтаксис регулярных выражений.
- Класс `Date` для представления и манипулирования значениями даты и времени.
- Класс `Error` и разнообразные его подклассы, экземпляры которых генерируются при возникновении ошибок в программах JavaScript.

[1] Не все, что здесь документируется, определено в спецификации языка JavaScript: некоторые описанные в главе классы и функции сначала были реализованы в веб-браузерах и затем приняты в Node, фактически превратившись в члены стандартной библиотеки JavaScript.

- Объект JSON, методы которого поддерживают сериализацию и десериализацию структур данных JavaScript, состоящих из объектов, массивов, строк, чисел и булевских значений.
- Объект Intl и определяемые им классы, которые могут помочь локализовать программы JavaScript.
- Объект Console, методы которого выводят строки способами, особенно полезными для отладки программ и регистрации поведения этих программ.
- Класс URL, который упрощает задачу разбора и манипулирования указателями URL. Кроме того, раскрываются также функции для кодирования и декодирования указателей URL и их составных частей.
- Функция setTimeout() и связанные функции для указания кода, подлежащего выполнению по истечении заданного промежутка времени.

Некоторые разделы в главе — в частности, разделы о типизированных массивах и регулярных выражениях — довольно длинные, поскольку до того, как вы сможете эффективно использовать такие типы, вам необходимо понять важную вводную информацию. Однако многие разделы будут короткими: в них просто представлен новый API-интерфейс и приведены примеры его применения.

11.1. Множества и отображения

Тип Object в JavaScript — это универсальная структура данных, которая может использоваться для отображения строк (имен свойств объекта) на произвольные значения. А когда отображаемое значение является чем-то фиксированным наподобие true, то объект фактически будет множеством строк.

На самом деле объекты применяются в качестве отображений и множеств при программировании на JavaScript достаточно регулярно. Тем не менее, все ограничивается строками и усложняется тем фактом, что объекты обычно наследуют свойства с именами вроде "toString", которые, как правило, не предназначены для того, чтобы быть частью отображения или множества.

По указанной причине в ES6 появились настоящие классы множеств Set и отображений Map, которые мы рассмотрим в последующих подразделах.

11.1.1. Класс Set

Подобно массиву множество является коллекцией значений. Однако в отличие от массивов множества не упорядочиваются, не индексируются и не допускают наличия дубликатов: значение либо будет членом множества, либо нет; запрашивать, сколько раз значение встречается внутри множества, невозможно.

Объект Set создается с помощью конструктора Set():

```
let s = new Set();          // Новое пустое множество
let t = new Set([1, s]);    // Новое множество с двумя членами
```

Аргумент конструктора `Set()` не обязан быть массивом: разрешен любой итерируемый объект (включая другие объекты `Set`):

```
let t = new Set(s);                    // Новое множество, которое
                                       // копирует элементы s.
let unique = new Set("Mississippi"); //4 элемента: "M", "i", "s" и "p"
```

Свойство `size` множества похоже на свойство `length` массива: оно сообщает нам, сколько значений содержит множество:

```
unique.size        // => 4
```

Множества не нуждаются в инициализации при их создании. Вы можете в любой момент добавлять и удалять элементы посредством `add()`, `delete()` и `clear()`. Вспомните, что множества не могут содержать дубликаты, поэтому добавление значения к множеству, когда такое значение уже содержится в нем, оказывается безрезультатным:

```
let s = new Set(); // Начать с пустого множества
s.size             // => 0
s.add(1);          // Добавить число
s.size             // => 1; теперь множество имеет один член
s.add(1);          // Добавить то же самое число еще раз
s.size             // => 1; размер не изменился
s.add(true);       // Добавить другое значение; обратите внимание
                   // на допустимость смешивания типов
s.size             // => 2
s.add([1,2,3]);    // Добавить значение типа массива
s.size             // => 3; был добавлен массив, а не его элементы
s.delete(1)        // => true: успешное удаление элемента 1
s.size             // => 2: размер уменьшился до 2
s.delete("test")   // => false: "test" не является членом,
                   // удаление терпит неудачу
s.delete(true)     // => true: удаление прошло успешно
s.delete([1,2,3])  // => false: массив во множестве отличается
s.size             // => 1: во множестве по-прежнему присутствует
                   // один массив
s.clear();         // Удалить все из множества
s.size             // => 0
```

Следует отметить несколько важных моментов, касающихся данного кода.

- Метод `add()` принимает единственный аргумент; если вы передадите массив, тогда `add()` добавит к множеству сам массив, а не индивидуальные элементы массива. Тем не менее, метод `add()` всегда возвращает множество, на котором вызывался, так что при желании добавить к множеству сразу несколько значений вы можете использовать выстроенные в цепочку вызовы метода вроде `s.add('a').add('b').add('c');`.
- Метод `delete()` тоже удаляет одиночный элемент множества за раз. Однако в отличие от `add()` метод `delete()` возвращает булевское значение. Если указанное вами значение действительно было членом множества, тогда `delete()` удаляет его и возвращает `true`. В противном случае он ничего не делает и возвращает `false`.

- Наконец, очень важно понимать, что членство во множестве основано на проверках на предмет строгого равенства как те, что выполняет операция `===`. Множество может содержать число 1 и строку `"1"`, потому что считает их несовпадающими значениями. Когда значения являются объектами (либо массивами или функциями), они также сравниваются, как если бы применялась операция `===`. Вот почему мы не смогли удалить элемент типа массива из множества в показанном выше коде. Мы добавили к множеству массив и затем попытались удалить его, передавая методу `delete()` другой массив (хотя с теми же самыми элементами). Чтобы все работало, нам нужно было передавать ссылку на точно тот же массив.

Примечание для программистов на Python: в этом заключается существенная разница между множествами JavaScript и Python. Множества Python сравнивают члены на предмет равенства, а не идентичности, но компромисс состоит в том, что во множествах Python разрешены только неизменяемые члены наподобие кортежей, и добавлять списки и словари к множествам не допускается.

На практике самое важное, что мы делаем с множествами — не добавление и удаление элементов из них, а проверка, является ли указанное значение членом множества. Для этого используется метод `has()`:

```
let oneDigitPrimes = new Set([2,3,5,7]);
oneDigitPrimes.has(2)      // => true: 2 - простое число с одной цифрой
oneDigitPrimes.has(3)      // => true: то же касается 3
oneDigitPrimes.has(4)      // => false: 4 - не простое число
oneDigitPrimes.has("5")    // => false: "5" - даже не число
```

Относительно множеств важнее всего понимать то, что они оптимизированы для проверки членства, и независимо от того, сколько членов содержится во множестве, метод `has()` будет очень быстрым. Метод `includes()` массива тоже выполняет проверку членства, но необходимое для проверки время пропорционально размеру массива, а применение массива в качестве множества может оказаться гораздо медленнее, чем использование настоящего объекта `Set`.

Класс `Set` является итерируемым, т.е. вы можете применять цикл `for/of` для прохода по всем элементам множества:

```
let sum = 0;
for(let p of oneDigitPrimes) {  // Пройти по простым числам с одной
   sum += p;                    // цифрой и сложить их
}
sum                             // => 17: 2 + 3 + 5 + 7
```

Поскольку объекты `Set` итерируемые, вы можете преобразовать их в массивы и списки аргументов с помощью операции распространения `...`:

```
[...oneDigitPrimes]             // => [2,3,5,7]: множество, преобразо-
                                //                ванное в Array
Math.max(...oneDigitPrimes)     // => 7: элементы множества,
                                //       передаваемые как аргументы
```

Множества часто описывают как "неупорядоченные коллекции". Тем не менее, это не совсем верно для класса Set в JavaScript. Множество JavaScript не поддерживает индексацию: вы не можете запрашивать, скажем, первый или третий элемент у множества, как могли бы делать в случае массива. Но класс Set в JavaScript всегда запоминает порядок, в котором вставлялись элементы, и он всегда использует такой порядок при проходе по множеству: первый вставленный элемент будет первым пройденным (при условии, что вы его сначала не удалили), а последний вставленный элемент — последним пройденным[2].

Помимо того, что класс Set является итерируемым, он также реализует метод forEach(), который похож на метод массива с таким же именем:

```
let product = 1;
oneDigitPrimes.forEach(n => { product *= n; });
product             // => 210: 2 * 3 * 5 * 7
```

Метод forEach() массива передает индексы массива в виде второго аргумента указанной вами функции. Множества не имеют индексов, а потому версия forEach() класса Set просто передает значение элемента как первый и второй аргументы.

11.1.2. Класс Map

Объект Map представляет набор значений, называемых *ключами*, где с каждым ключом ассоциировано (или "отображено" на него) еще одно значение. В некотором смысле отображение похоже на массив, но вместо применения набора последовательных целых чисел для ключей отображения позволяют использовать в качестве "индексов" произвольные значения. Подобно массивам отображения характеризуются высокой скоростью работы: поиск значения, ассоциированного с ключом, будет быстрым (хотя и не настолько быстрым, как индексация в массиве) независимо от размера отображения.

Новое отображение создается с помощью конструктора Map():

```
let m = new Map();  // Новое пустое отображение
let n = new Map([   // Новое отображение, которое инициализировано
                    // строковыми ключами, отображенными на числа
    ["one", 1],
    ["two", 2]
]);
```

Необязательным аргументом конструктора Map() должен быть итерируемый объект, который выдает массивы из двух элементов [ключ, значение]. На практике это означает, что если вы хотите инициализировать отображение при его создании, то обычно будете записывать желаемые ключи и ассоциированные с ними значения в виде массива массивов. Но вы также можете применять конструктор Map() для копирования других отображений или имен и значений свойств из существующего объекта:

[2] Такой предсказуемый порядок итерации — еще один аспект, касающийся множеств JavaScript, который программисты на Python могут счесть удивительным.

```
let copy = new Map(n);  // Новое отображение с такими же ключами
                        // и значениями, как у отображения n
let o = { x: 1, y: 2};  // Объект с двумя свойствами
let p = new Map(Object.entries(o)); // То же, что и
                                    // new map([["x", 1], ["y", 2]])
```

Создав объект Map, вы можете запрашивать значение, ассоциированное с заданным ключом, с помощью `get()` и добавлять новую пару ключ/значение посредством `set()`. Однако не забывайте, что отображение представляет собой набор ключей, каждый из которых имеет ассоциированное значение. Оно не полностью аналогично набору пар ключ/значение. Если вы вызовите `set()` с ключом, который уже существует в отображении, то измените значение, ассоциированное с этим ключом, а не добавите новое сопоставление ключ/значение. В дополнение к `get()` и `set()` класс Map также определяет методы, похожие на методы класса Set: используйте `has()` для проверки, включает ли отображение указанный ключ; применяйте `delete()` для удаления ключа (и ассоциированного с ним значения) из отображения; используйте `clear()` для удаления всех пар ключ/значение из отображения; применяйте свойство `size` для выяснения, сколько ключей содержит отображение.

```
let m = new Map(); // Начать с пустого отображения
m.size             // => 0: пустые отображения вообще не содержат ключей
m.set("one", 1);   // Отобразить ключ "one" на значение 1
m.set("two", 2);   // Отобразить ключ "two" на значение 2
m.size             // => 2: теперь отображение имеет два ключа
m.get("two")       // => 2: возвращается значение, ассоциированное с
ключом "two"
m.get("three")     // => undefined: этот ключ отсутствует в отображении
m.set("one", true); // Изменить значение, ассоциированное
                   // с существующим ключом
m.size             // => 2: размер не изменился
m.has("one")       // => true: отображение имеет ключ "one"
m.has(true)        // => false: отображение не содержит ключа true
m.delete("one")    // => true: ключ существует, и удаление
                   //          проходит успешно
m.size             // => 1
m.delete("three")  // => false: не удалось удалить несуществующий ключ
m.clear();         // Удалить все ключи и значения из отображения
```

Подобно методу `add()` класса Set вызовы метода `set()` класса Map можно выстраивать в цепочки, что позволяет инициализировать отображения, не используя массивы массивов:

```
let m = new Map().set("one", 1).set("two", 2).set("three", 3);
m.size             // => 3
m.get("two")       // => 2
```

Как и в Set, в качестве ключа или значения Map можно применять любое значение JavaScript, включая `null`, `undefined` и `NaN`, а также ссылочные типы вроде объектов и массивов. И подобно классу Set класс Map сравнивает ключи на предмет идентичности, а не равенства, так что если вы используете для ключа

объект или массив, то он будет считаться отличающимся от любого другого объекта или массива даже при наличии в нем одинаковых свойств или элементов:

```
let m = new Map();          // Начать с пустого отображения
m.set({}, 1);               // Отобразить один пустой объект на число 1
m.set({}, 2);               // Отобразить другой пустой объект на число 2
m.size                      // => 2: в отображении есть два ключа
m.get({})    // => undefined: но этот пустой объект не является ключом
m.set(m, undefined);        // Отобразить само отображение
                            // на значение undefined
m.has(m)                    // => true: отображение m - ключ в самом себе
m.get(m)                    // => undefined: такое же значение мы получили
                            //                 бы, если бы m не было ключом
```

Объекты Map являются итерируемыми, причем каждое итерируемое значение представляет собой двухэлементный массив, где первый элемент — ключ, а второй — ассоциируемое с ключом значение. Если вы примените операцию распространения к объекту Map, то получите массив массивов вроде того, что передавался конструктору Map(). При проходе по отображению с помощью цикла for/of идиоматично использовать деструктурирующее присваивание, чтобы присвоить ключ и значение отдельным переменным:

```
let m = new Map([["x", 1], ["y", 2]]);
[...m]    // => [["x", 1], ["y", 2]]

for(let [key, value] of m) {
   // На первой итерации ключом будет "x", а значением - 1
   // На второй итерации ключом будет "y", а значением - 2
}
```

Как и Set, класс Map выполняет проход в порядке вставки. Во время итерации первой парой ключ/значение будет та, которая добавлялась в отображение раньше остальных, а последней парой — та, что добавлялась позже остальных.

Если вы хотите проходить только по ключам или только по ассоциированным значениям отображения, тогда применяйте методы keys() и values(): они возвращают итерируемые объекты, которые обеспечивают проход по ключам и значениям в порядке вставки. (Метод entries() возвращает итерируемый объект, который делает возможным проход по парам ключ/значение, но результат будет в точности таким же, как итерация непосредственно по отображению.)

```
[...m.keys()]           // => ["x", "y"]: только ключи
[...m.values()]         // => [1, 2]: только значения
[...m.entries()]        // => [["x", 1], ["y", 2]]: то же, что и [...m]
```

Объекты Map можно подвергать итерации также с использованием метода forEach(), который впервые был реализован в классе Array:

```
m.forEach((value, key) => {  // обратите внимание на указание value,
                             // key, а НЕ key, value
   // На первой итерации значением будет 1, а ключом - "x"
   // На второй итерации значением будет 2, а ключом - "y"
});
```

В приведенном выше коде может показаться странным, что параметр значения value находится перед параметром ключа key, поскольку в итерации for/of ключ следует первым. Как отмечалось в начале раздела, вы можете считать отображение обобщенным массивом, в котором целочисленные индексы массива заменены произвольными значениями ключей. Методу forEach() массивов элемент массива передается первым, а индекс массива — вторым, так что по аналогии методу forEach() отображения значение отображения передается первым, а ключ отображения — вторым.

11.1.3. WeakMap и WeakSet

Класс WeakMap является разновидностью (но не фактическим подклассом) класса Map, которая не препятствует обработке значений своих ключей сборщиком мусора. Сборка мусора — это процесс, посредством которого интерпретатор JavaScript восстанавливает память, занимаемую объектами, которые перестали быть "достижимыми" и не могут использоваться программой. Обыкновенное отображение хранит "сильные" ссылки на значения своих ключей, и они остаются достижимыми через отображение, даже если все прочие ссылки на них исчезли. Напротив, WeakMap хранит "слабые" ссылки на значения своих ключей, так что они не достижимы через WeakMap, а их присутствие в отображении не препятствует восстановлению занимаемой ими памяти.

Конструктор WeakMap() похож на конструктор Map(), но между WeakMap и Map существует несколько важных отличий.

- Ключи WeakMap обязаны быть объектами или массивами; элементарные значения не подлежат сборке мусора и не могут применяться в качестве ключей.
- Класс WeakMap реализует только методы get(), set(), has() и delete(). В частности, WeakMap не является итерируемым и не определяет keys(), values() или forEach(). Если бы класс WeakMap был итерируемым, тогда его ключи оказались бы достижимыми и не слабыми.
- Аналогично WeakMap не реализует свойство size, потому что размер WeakMap может измениться в любой момент из-за обработки объектов сборщиком мусора.

Класс WeakMap задумывался для того, чтобы дать возможность ассоциировать значения с объектами, не вызывая утечек памяти. Например, пусть вы пишете функцию, которая принимает объектный аргумент и должна выполнить отнимающее много времени вычисление над этим объектом. Для повышения эффективности вы хотите кешировать вычисленное значение для использования в будущем. Если вы реализуете кеш с использованием объекта Map, то воспрепятствуете восстановлению памяти, занимаемой любым объектом, но в случае применения WeakMap проблема не возникнет. (Часто вы можете добиться похожего результата, используя закрытое свойство Symbol для кеширования вычисленного значения прямо в объекте. См. подраздел 6.10.3.)

Класс `WeakSet` реализует множество объектов, которые не препятствуют обработке этих объектов сборщиком мусора. Конструктор `WeakSet()` работает подобно конструктору `Set()`, но объекты `WeakSet` отличаются от объектов `Set` в тех же аспектах, в каких объекты `WeakMap` отличаются от объектов `Map`.

- Класс `WeakSet` не разрешает элементарным значениям быть членами.
- Класс `WeakSet` реализует только методы `add()`, `has()` и `delete()` и не является итерируемым.
- Класс `WeakSet` не имеет свойства `size`.

Класс `WeakSet` применяется нечасто: его сценарии использования похожи на подобные сценарии для `WeakMap`. Если вы хотите пометить (или "маркировать") объект как имеющий какое-то особое свойство или тип, например, тогда можете добавить его в `WeakSet`. Затем где-то в другом месте, когда вы пожелаете выяснить наличие у объекта такого свойства или типа, то можете проверить его членство в `WeakSet`. Работа с обыкновенным множеством помешала бы обработке всех помеченных объектов сборщиком мусора, но это не проблема в случае применения `WeakSet`.

11.2. Типизированные массивы и двоичные данные

Обыкновенные массивы JavaScript могут содержать элементы любого типа и способны динамически увеличиваться или уменьшаться. Реализации JavaScript проводят многочисленные оптимизации, а потому обычные случаи использования массивов JavaScript характеризуются очень высокой скоростью. Тем не менее, они по-прежнему сильно отличаются от типов массивов языков более низкого уровня вроде C и Java. *Типизированные массивы*, появившиеся в ES6,[3] намного ближе к низкоуровневым массивам в упомянутых языках. Типизированные массивы формально не являются массивами (`Array.isArray()` возвращает для них `false`), но они реализуют все методы массивов, описанные в разделе 7.8, плюс несколько собственных методов. Однако типизированные массивы отличаются от обыкновенных массивов в нескольких важных аспектах.

- Все элементы типизированного массива представляют собой числа. Тем не менее, в отличие от обыкновенных чисел JavaScript типизированные массивы позволяют указывать тип (целочисленный со знаком и без знака и с плавающей точкой IEEE-754) и размер (от 8 до 64 бит) чисел, которые будут храниться в массиве.
- Вы должны указывать длину типизированного массива при его создании, и эта длина никогда не изменяется.
- Элементы типизированного массива всегда инициализируются 0 при создании массива.

[3] Типизированные массивы впервые появились в JavaScript на стороне клиента, когда в веб-браузеры была добавлена поддержка графики WebGL. Нововведением ES6 стало то, что они были подняты до уровня базового языкового средства.

11.2.1. Типы типизированных массивов

В JavaScript отсутствует класс TypedArray. Взамен есть 11 видов типизированных массивов, каждый с отличающимся типом элементов и конструктором (табл. 11.1).

Таблица 11.1. Виды типизированных массивов

Конструктор	Числовой тип
`Int8Array()`	байты со знаком
`Uint8Array()`	байты без знака
`Uint8ClampedArray()`	байты без знака и без переноса бит
`Int16Array()`	16-битные короткие целые числа со знаком
`Uint16Array()`	16-битные короткие целые числа без знака
`Int32Array()`	32-битные целые числа со знаком
`Uint32Array()`	32-битные целые числа без знака
`BigInt64Array()`	64-битные значения `BigInt` со знаком (ES2020)
`BigUint64Array()`	64-битные значения `BigInt` без знака (ES2020)
`Float32Array()`	32-битные значения с плавающей точкой
`Float64Array()`	64-битные значения с плавающей точкой: обыкновенные числа JavaScript

Типы с именами, начинающимися на `Int`, хранят целые числа со знаком, состоящие из 1, 2 или 4 байтов (8, 16 или 32 бит). Типы, имена которых начинаются с `Uint`, хранят целые числа без знака той же длины. Типы с именами, начинающимися на `BigInt` и `BigUint`, хранят 64-битные целые числа, представляемые в JavaScript как значения `BigInt` (см. подраздел 3.2.5). Типы, имена которых начинаются с `Float`, хранят числа с плавающей точкой. Элементы массива `Float64Array` имеют тот же тип, что и обыкновенные числа JavaScript. Элементы массива `Float32Array` обладают более низкой точностью и меньшим диапазоном, но требуют только половину памяти. (В языках C и Java такой тип называется `float`.)

Тип `Uint8ClampedArray` является вариантом особого случая типа `Uint8Array`. Оба типа хранят байты без знака и могут представлять числа между 0 и 255. При сохранении в элементе массива `Uint8Array` значения больше 255 или меньше 0 оно "заворачивается" и вы получаете какую-то другую величину. Именно так работает компьютерная память на низком уровне, поэтому все происходит очень быстро. В `Uint8ClampedArray` предпринимается дополнительная проверка типа, так что если вы сохраняете значение больше 255 или меньше 0, то оно "фиксируется" в 255 или 0 и не заворачивается. (Такое поведение фиксации затребовано низкоуровневым API-интерфейсом HTML-элемента `<canvas>` для манипулирования цветами пикселей.)

Каждый конструктор типизированного массива имеет свойство `BYTES_PER_ELEMENT` со значением 1, 2, 4 или 8 в зависимости от типа.

11.2.2. Создание типизированных массивов

Простейший способ создания типизированного массива предусматривает вызов подходящего конструктора с одним числовым аргументом, который задает количество элементов в массиве:

```
let bytes = new Uint8Array(1024);     // 1024 байта
let matrix = new Float64Array(9);     // Матрица 3x3
let point = new Int16Array(3);        // Точка в трехмерном пространстве
let rgba = new Uint8ClampedArray(4); // 4-байтовое значение RGBA пикселя
let sudoku = new Int8Array(81);       // Доска для игры в судоку 9x9
```

Когда вы создаете типизированный массив подобным образом, все элементы массива гарантированно инициализируются значением 0, 0n или 0.0. Но если вам известны значения, которые желательно поместить в типизированный массив, тогда вы можете также указать их при создании массива. Каждый конструктор типизированного массива имеет статические фабричные методы `from()` и `of()`, которые работают аналогично методам `Array.from()` и `Array.of()`:

```
let white = Uint8ClampedArray.of(255, 255, 255, 0); // Непрозрачный
                                                    // белый цвет RGBA
```

Вспомните, что фабричный метод `Array.from()` ожидает в своем первом аргументе объект, похожий на массив, или итерируемый объект. То же самое справедливо для разновидностей типизированных массивов за исключением того, что итерируемый объект или объект, похожий на массив, также обязан иметь числовые элементы. Скажем, строки итерируемы, но их не имеет смысла передавать фабричному методу `from()` типизированного массива.

Если вы просто применяете версию `from()` с одним аргументом, то можете отбросить `.from` и передавать свой итерируемый объект или объект, похожий на массив, напрямую функции конструктора, что обеспечивает в точности то же самое поведение. Обратите внимание, что конструктор и фабричный метод `from()` позволяют копировать существующие типизированные массивы, одновременно допуская изменение типа:

```
let ints = Uint32Array.from(white); // Те же 4 числа, но как целые
```

При создании нового типизированного массива из существующего массива, итерируемого объекта или объекта, похожего на массив, значения могут усекаться, чтобы удовлетворять ограничениям типа создаваемого массива. Когда это происходит, никакие предупреждающие сообщения или сообщения об ошибках не выдаются:

```
// Числа с плавающей точкой усекаются до целых, более длинные
// целые усекаются до 8 бит
Uint8Array.of(1.23, 2.99, 45000)    // => new Uint8Array([1, 2, 200])
```

Наконец, есть еще один способ создания типизированных массивов, который задействует тип `ArrayBuffer` — непрозрачную ссылку на порцию памяти. Вы можете создать экземпляр `ArrayBuffer` с помощью конструктора; просто передайте количество байтов памяти для выделения:

```
let buffer = new ArrayBuffer(1024*1024);
buffer.byteLength  // => 1024*1024; один мегабайт памяти
```

Класс ArrayBuffer не позволяет читать или записывать любые байты, которые были выделены. Но вы можете создавать типизированные массивы, которые используют память буфера и дают возможность читать и записывать в нее. Для этого вызовите конструктор типизированного массива с экземпляром ArrayBuffer в первом аргументе, байтовым смещением внутри буфера массива во втором аргументе и длиной массива (в элементах, не в байтах) в третьем аргументе. Второй и третий аргументы необязательны. Если вы опустите их обоих, то массив будет работать со всей памятью в буфере массива. Если вы опустите только аргумент длины, тогда массив будет потреблять все доступную память между начальной позицией и концом массива. Есть еще один момент, касающийся этой формы конструктора типизированного массива, о котором следует помнить: массивы обязаны быть выровненными в памяти, а потому в случае указания байтового смещения его значение должно быть кратным размеру применяемого типа. Например, для конструктора Int32Array() требуется число, кратное 4, а для конструктора Float64Array() — число, кратное 8.

Располагая ранее созданным экземпляром ArrayBuffer, вот как можно было бы создать типизированные массивы:

```
let asbytes = new Uint8Array(buffer); // Представление в виде байтов
let asints = new Int32Array(buffer);  // Представление в виде 32-
                                      // битных целых чисел со знаком
let lastK = new Uint8Array(buffer, 1023*1024);  // Последний килобайт
                                                // как байты
let ints2 = new Int32Array(buffer, 1024, 256);  // Второй килобайт как
                                                // 256 целых чисел
```

Четыре созданных типизированных массива предлагают четыре разных представления памяти, доступной через ArrayBuffer. Важно понимать, что все типизированные массивы имеют лежащий в основе объект ArrayBuffer, даже когда он явно не указан. В случае вызова конструктора типизированного массива без передачи ему объекта буфера автоматически создается буфер подходящего размера. Как будет объясняться позже, свойство buffer любого типизированного массива ссылается на внутренний объект ArrayBuffer. Причина работы напрямую с объектами ArrayBuffer связана с тем, что временами желательно иметь несколько представлений в виде типизированных массивов одного буфера.

11.2.3. Использование типизированных массивов

После создания типизированного массива вы можете читать и записывать его элементы с помощью обычной формы с квадратными скобками, как поступали бы с любым другим объектом, похожим на массив:

```
// Возвращает наибольшее простое число меньше n,
// используя решето Эратосфена
function sieve(n) {
    let a = new Uint8Array(n+1);  // a[x] будет 1, если x сложное число
    let max = Math.floor(Math.sqrt(n));  // Множители не должны быть
                                         // больше этого
```

```
    let p = 2;                          // 2 - первое простое число
    while(p <= max) {                   // Для простых чисел меньше max
      for(let i = 2*p; i <= n; i += p)  // Пометить числа, кратные p,
                                        // как сложные
        a[i] = 1;
      while(a[++p]) /* пустое тело */; //Следующий непомеченный индекс
                                        //является простым числом
    }
    while(a[n]) n--;            // Цикл в обратном направлении для поиска
                                // последнего простого числа
    return n;                   // Возвращение найденного простого числа
}
```

Функция `sieve()` находит наибольшее простое число, которое меньше переданного ей числа. Код точно такой же, как если бы работа велась с обыкновенным массивом JavaScript, но согласно моему тестированию применение `Uint8Array()` вместо `Array()` повысило скорость выполнения кода более чем в четыре раза и сократило потребление памяти в восемь раз.

Типизированные массивы — не подлинные массивы, но они реализуют большинство методов массивов, поэтому вы можете использовать их почти так же, как обычные массивы:

```
let ints = new Int16Array(10);       // 10 коротких целых чисел
ints.fill(3).map(x=>x*x).join("")    // => "9999999999"
```

Не забывайте, что типизированные массивы имеют фиксированные длины, поэтому свойство `length` допускает только чтение, а методы, изменяющие длину массива (такие как `push()`, `pop()`, `unshift()`, `shift()` и `splice()`), не реализованы. С другой стороны, методы, которые изменяют содержимое массива, не меняя его длину (вроде `sort()`, `reverse()` и `fill()`), реализованы. Методы наподобие `map()` и `slice()`, возвращающие новые массивы, возвращают типизированный массив того же типа, на котором они были вызваны.

11.2.4. Методы и свойства типизированных массивов

В дополнение к стандартным методам массивов типизированные массивы реализуют несколько собственных методов. Метод `set()` устанавливает множество элементов типизированного массива одновременно, копируя в него элементы обыкновенного или типизированного массива:

```
let bytes = new Uint8Array(1024);          // Буфер размером 1 килобайт
let pattern = new Uint8Array([0,1,2,3]); // Массив из 4 байтов
bytes.set(pattern); //Копировать их в начало другого байтового массива
bytes.set(pattern, 4);      // Копировать их снова по другому смещению
bytes.set([0,1,2,3], 8);    // Или просто копировать значения прямо
                            // из обыкновенного массива
bytes.slice(0, 12)  // => new Uint8Array([0,1,2,3,0,1,2,3,0,1,2,3])
```

Метод `set()` получает в своем первом аргументе массив или типизированный массив и в необязательном втором аргументе смещение элемента, которое по умолчанию принимается равным 0, когда не указано. Операция копирования

значений из одного типизированного массива в другой, по всей видимости, будет чрезвычайно быстрой.

Типизированные массивы также имеют метод subarray(), возвращающий часть массива, на котором он вызван:

```
let ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]);     // 10 коротких
                                                      // целых чисел
let last3 = ints.subarray(ints.length-3, ints.length);   // Последние
                                                         // 3 из них
last3[0]          // => 7: то же, что и ints[7]
```

Метод subarray() принимает такие же аргументы, как метод slice(), и выглядит работающим аналогично. Но имеется важное отличие. Метод slice() возвращает указанные элементы в новом независимом типизированном массиве, который не разделяет память с первоначальным массивом. Метод subarray() не копирует данные в памяти; он просто возвращает новое представление тех же самых лежащих в основе значений:

```
ints[9] = -1;       // Изменяет значение в первоначальном массиве и...
last3[2]            // => -1: также изменяет значение в подмассиве
```

Тот факт, что метод subarray() возвращает новое представление существующего массива, снова поднимает тему объектов ArrayBuffer. Каждый типизированный массив имеет три свойства, относящиеся к внутренним буферам:

```
last3.buffer        // Объект ArrayBuffer для типизированного массива
last3.buffer === ints.buffer  // => true: оба являются представлениями
                              // того же самого буфера
last3.byteOffset   //=>14:это представление начинается с байта 14 буфера
last3.byteLength   // => 6: это представление имеет длину 6 байтов
                   //       (три 16-битных целых)
last3.buffer.byteLength //=>20: но внутренний буфер содержит 20 байтов
```

Свойство buffer — это объект ArrayBuffer массива. Свойство byteOffset — начальная позиция данных массива внутри лежащего в основе буфера. Свойство byteLength — длина данных массива в байтах. Для любого типизированного массива a следующий инвариант всегда должен быть истинным:

```
a.length * a.BYTES_PER_ELEMENT === a.byteLength  // => true
```

Объекты ArrayBuffer — всего лишь непрозрачные порции байтов. Вы можете получать доступ к этим байтам с помощью типизированных массивов, но сам объект ArrayBuffer не является типизированным массивом. Однако будьте осторожны: вы можете применять с объектами ArrayBuffer числовую индексацию массивов, как в отношении любого объекта JavaScript. Это не откроет вам доступ к байтам в буфере, но может стать причиной сбивающих с толку ошибок:

```
let bytes = new Uint8Array(8);
bytes[0] = 1;             // Установить первый байт в 1
bytes.buffer[0]           // => undefined: буфер не имеет индекс 0
bytes.buffer[1] = 255; //Попытаться некорректно установить байт в буфере
```

```
bytes.buffer[1]    // => 255: это просто устанавливает обыкновенное
                   //         свойство объекта JavaScript
bytes[1]           // => 0: строка кода выше не установили байт
```

Ранее было показано, что вы можете создавать объект `ArrayBuffer` посредством конструктора `ArrayBuffer()` и затем создавать типизированные массивы, которые используют этот буфер. Еще один подход предусматривает создание начального типизированного массива и применение его буфера для создания других представлений:

```
let bytes = new Uint8Array(1024);              // 1024 байта
let ints = new Uint32Array(bytes.buffer);      // или 256 целых чисел
let floats = new Float64Array(bytes.buffer); // или 128 чисел
                                               // с плавающей точкой
```

11.2.5. DataView и порядок байтов

Типизированные массивы позволяют представлять одну и ту же последовательность байтов с помощью порций из 8, 16, 32 или 64 бит. Это демонстрирует "порядок байтов", в котором байты организованы в более длинные слова. Типизированные массивы ради эффективности используют естественный порядок байтов лежащего в основе оборудования. В системах с порядком от младшего к старшему байты числа в `ArrayBuffer` располагаются от наименее значащего до наиболее значащего. В системах с порядком от старшего к младшему байты располагаются от наиболее значащего до наименее значащего. Вы можете выяснить порядок байтов платформы посредством кода следующего вида:

```
// Если целое число 0x00000001 размещено в памяти как 01 00 00 00,
// тогда мы работаем на платформе с порядком от младшего к старшему.
// В случае платформы с порядком от старшего к младшему мы взамен
// получили бы байты 00 00 00 01.
let littleEndian = new Int8Array(new Int32Array([1]).buffer)[0] === 1;
```

В наши дни наиболее всего распространены архитектуры центральных процессоров с порядком от младшего к старшему. Тем не менее, многие сетевые протоколы и некоторые форматы двоичных файлов требуют порядка от старшего к младшему. В случае применения типизированных массивов с данными, которые поступают из сети или файла, вы не можете предполагать, что порядок байтов платформы будет совпадать с порядком байтов данных. В целом при работе с внешними данными вы можете использовать классы `Int8Array` и `Uint8Array` для представления данных в виде массива индивидуальных байтов, но не следует применять остальные типизированные массивы с многобайтовыми размерами слов. Взамен вы можете использовать класс `DataView`, который определяет методы для чтения и записи значений из `ArrayBuffer` с явно указываемым порядком байтов:

```
// Предположим, что у нас есть типизированный массив байтов
// двоичных данных для обработки.
// Первым делом мы создаем объект DataView, чтобы можно было гибким
// образом читать и записывать значения из этих байтов.
```

```
let view = new DataView(bytes.buffer,
                        bytes.byteOffset,
                        bytes.byteLength);
let int = view.getInt32(0); // Прочитать целое число со знаком и порядком
                            // от старшего к младшему из байта 0.
int = view.getInt32(4, false);  // Следующее целое число тоже имеет
                                // порядок от старшего к младшему.
int = view.getUint32(8, true);  // Следующее целое число без знака
                                //имеет порядок от младшего к старшему.
view.setUint32(8, int, false); // Записать его в формате с порядком
                               // от старшего к младшему.
```

Класс DataView определяет десять методов получения для десяти классов типизированных массивов (исключая Uint8ClampedArray), такие как getInt16(), getUint32(), getBigInt64() и getFloat64(). В первом аргументе указывается байтовое смещение внутри ArrayBuffer, с которого начинается значение. Все эти методы получения кроме getInt8() и getUint8() принимают в необязательном втором аргументе булевское значение. Если второй аргумент опущен или равен false, то применяется порядок от старшего к младшему. Если второй аргумент равен true, тогда используется порядок от младшего к старшему.

Класс DataView также определяет десять методов установки, которые записывают значения во внутренний объект ArrayBuffer. В первом аргументе передается смещение, с которого начинается значение, а во втором аргументе — записываемое значение. Каждый метод кроме setInt8() и setUint8() принимает необязательный третий аргумент. Если он опущен или равен false, то значение записывается в формате с порядком от старшего к младшему, с наиболее значащим байтом в начале. Если третий аргумент равен true, тогда значение записывается в формате с порядком от младшего к старшему, с наименее значащим байтом в начале.

Типизированные массивы и класс DataView предоставляют вам все инструменты, необходимые для обработки двоичных данных, и позволяют писать программы JavaScript, которые выполняют такие действия, как распаковка файлов ZIP либо извлечение метаданных из файлов JPEG.

11.3. Сопоставление с образцом с помощью регулярных выражений

Регулярное выражение — это объект, который описывает текстовый шаблон. Регулярные выражения в JavaScript представляются посредством класса RegExp, а в классах String и RegExp определены методы, которые применяют регулярные выражения для выполнения мощных функций сопоставления с образцом и поиска с заменой в тексте. Однако для эффективного использования API-интерфейса RegExp вы должны также научиться описывать шаблоны текста с применением грамматики регулярных выражений, которая по существу сама по себе является мини-языком программирования. К счастью, грамматика ре-

гулярных выражений JavaScript очень похожа на грамматику, используемую во многих других языках программирования, так что вам она уже может быть знакома. (Если же нет, то изучение регулярных выражений JavaScript с высокой вероятностью принесет пользу также и в других контекстах программирования.)

В последующих подразделах сначала описывается грамматика регулярных выражений, а затем после объяснения, как писать регулярные выражения, будет демонстрироваться их применение с методами классов `String` и `RegExp`.

11.3.1. Определение регулярных выражений

Регулярные выражения в JavaScript представляются объектами `RegExp`. Конечно же, объекты `RegExp` можно создавать с помощью конструктора `RegExp()`, но чаще они создаются с использованием специального литерального синтаксиса. Подобно тому, как строковые литералы указываются в виде символов внутри кавычек, литералы регулярных выражений указываются в виде символов внутри пары символов обратной косой черты (/), а потому код JavaScript может содержать строки вроде:

```
let pattern = /s$/;
```

Приведенная выше строка создает новый объект `RegExp` и присваивает его переменной `pattern`. Созданный объект `RegExp` соответствует любой строке, которая заканчивается буквой "s". То же самое регулярное выражение можно было бы создать посредством конструктора `RegExp()`:

```
let pattern = new RegExp("s$");
```

Спецификации шаблонов регулярных выражений состоят из последовательности символов. Большинство символов, включая все алфавитно-цифровые символы, просто описывают символы для буквального сопоставления. Таким образом, регулярное выражение `/java/` соответствует любой строке, которая содержит подстроку "java". Другие символы в регулярных выражениях не сопоставляются буквально, но имеют особый смысл. Например, регулярное выражение `/s$/` содержит два символа. Первый, "s", сопоставляется с собой буквально. Второй, "$", является метасимволом, который обозначает конец строки. В итоге это регулярное выражение соответствует любой строке, которая содержит букву "s" в качестве своего последнего символа.

Как будет показано, регулярные выражения могут также содержать один или большее количество флагов, которые влияют на их работу. Флаги указываются после второго символа обратной косой черты в литералах `RegExp` или во втором строковом аргументе конструктора `RegExp()`. Скажем, если нужно соответствие строкам, которые заканчиваются символом "s" или "S", то с нашим регулярным выражением можно применить флаг `i` для указания на то, что нас интересует сопоставление, нечувствительное к регистру:

```
let pattern = /s$/i;
```

Далее описаны различные символы и метасимволы, используемые в регулярных выражениях JavaScript.

Буквальные символы

Все алфавитные символы и цифры в регулярных выражениях сопоставляются сами с собой буквально. Синтаксис регулярных выражений JavaScript также поддерживает определенные неалфавитные символы через управляющие последовательности, которые начинаются с символа обратной косой черты (\). Например, последовательность \n соответствует символу новой строки. Такие символы перечислены в табл. 11.2.

Таблица 11.2. Буквальные символы в регулярных выражениях

Символ	Соответствие
Алфавитно-цифровой символ	Сам себе
\0	Символ NUL (\u0000)
\t	Табуляция (\u0009)
\n	Новая строка (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\x*nn*	Символ Latin, указанный шестнадцатеричным числом *nn*; например, \x0A — то же самое, что и \n
\u*xxxx*	Символ Unicode, указанный шестнадцатеричным числом *xxxx*; например, \u0009 — то же самое, что и \t
\u{*n*}	Символ Unicode, указанный кодовой точкой *n*, где *n* — от одной до шести шестнадцатеричных цифр между 0 и 10FFFF. Обратите внимание, что такой синтаксис поддерживается только в регулярных выражениях, использующих флаг u
\c*X*	Управляющий символ ^*X*; например, \cJ эквивалентно символу новой строки \n

Несколько символов пунктуации имеют особый смысл в регулярных выражениях. Вот они:

```
^ $ . * + ? = ! : | \ / ( ) [ ] { }
```

Смысл этих символов обсуждается далее в главе. Некоторые из них имеют особый смысл лишь внутри определенных контекстов регулярного выражения, а в остальных контекстах трактуются буквально. Тем не менее, запомните в качестве общего правила: если вы хотите включить в регулярное выражение любой из перечисленных выше символов пунктуации буквально, то должны предварить его символом \. Другие символы пунктуации, такие как кавычки и @, не имеют особого смысла и просто сопоставляются в регулярном выражении сами с собой буквально.

Если вам не удается вспомнить, какие символы пунктуации необходимо отменять с помощью обратной косой черты, тогда можете благополучно помещать

обратную косую черту перед любым символом пунктуации. С другой стороны, обратите внимание, что многие буквы и цифры имеют особый смысл, когда предваряются обратной косой чертой, поэтому любые буквы и цифры, которые желательно сопоставлять буквально, не должны отменяться посредством обратной косой черты. Чтобы включить символ обратной косой черты буквальным образом в регулярное выражение, его потребуется отменить с помощью обратной косой черты. Скажем, следующее регулярное выражение соответствует любой строке, содержащей обратную косую черту: `/\\/`. (И если вы применяете конструктор `RegExp()`, то имейте в виду, что любые символы обратной косой черты в регулярном выражении должны быть продублированы, поскольку строки также используют обратную косую черту в качестве управляющего символа.)

Классы символов

Отдельные буквальные символы можно объединять в *классы символов*, помещая их внутрь квадратных скобок. Класс символов соответствует любому содержащемуся в нем символу. Таким образом, регулярное выражение `/[abc]/` соответствует любой из букв a, b или c. Можно также определять классы символов с отрицанием; они соответствуют любому символу, исключая те, что содержатся внутри квадратных скобок. Класс символов с отрицанием указывается с применением знака вставки (^) в качестве первого символа внутри квадратных скобок. Объект `RegExp` вида `/[^abc]/` соответствует любому одному символу, отличающемуся от a, b и c. Классы символов могут использовать дефис для указания диапазона символов. Чтобы соответствовать любому одному символу нижнего регистра из алфавита Latin, применяйте `/[a-z]/`, а для соответствия любой букве или цифре из алфавита Latin используйте `/[a-zA-Z0-9]/`. (И если вы хотите включить фактический дефис в свой класс символов, тогда сделайте его последним символом перед правой квадратной скобкой.)

Из-за того, что определенные классы символов применяются достаточно часто, в синтаксисе регулярных выражений JavaScript предусмотрены специальные символы и управляющие последовательности для представления таких распространенных классов. Например, `\s` соответствует символу пробела, символу табуляции и любому другому пробельному символу Unicode; `\S` соответствует любому символу, который *не* является пробельным символом Unicode. В табл. 11.3 перечислены эти символы и подытожен синтаксис классов символов. (Обратите внимание, что некоторые управляющие последовательности классов символов соответствуют только символам ASCII и не были расширены для работы с символами Unicode. Однако вы можете явно определить собственные классы символов Unicode; скажем, `/[\u0400-\u04FF]/` соответствует любому одному кириллическому символу.)

Обратите внимание, что специальные управляющие последовательности классов символов могут применяться внутри квадратных скобок. `\s` соответствует любому пробельному символу, а `\d` — любой цифре, поэтому `/[\s\d]/` соответствует любому одному пробельному символу или цифре.

Таблица 11.3. Классы символов в регулярных выражениях

Класс символов	Соответствие
`[...]`	Любой один символ, находящийся между квадратными скобками
`[^...]`	Любой один символ, не находящийся между квадратными скобками
`.`	Любой символ за исключением символа новой строки или другого разделителя строк Unicode. Или если RegExp использует флаг `s`, тогда точка соответствует любому символу, в том числе разделителям строк
`\w`	Любой символ слов ASCII. Эквивалентно `[a-zA-Z0-9_]`
`\W`	Любой символ, который не является символом слов ASCII. Эквивалентно `[^a-zA-Z0-9_]`
`\s`	Любой пробельный символ Unicode
`\S`	Любой символ, не являющийся пробельным символом Unicode
`\d`	Любая цифра ASCII. Эквивалентно `[0-9]`
`\D`	Любой символ, отличающийся от цифры ASCII. Эквивалентно `[^0-9]`
`[\b]`	Буквальный забой (особый случай)

Следует отметить, что существует один особый случай. Как вы увидите позже, управляющая последовательность `\b` имеет особый смысл. Тем не менее, когда она используется внутри класса символов, то представляет символ забоя. Таким образом, для буквального представления символа забоя в регулярном выражении применяйте класс символов с одним элементом: `/[\b]/`.

Классы символов Unicode

Если регулярное выражение в ES2018 использует флаг `u`, тогда поддерживаются классы символов `\p{...}` и их отрицание `\P{...}`. (По состоянию на начало 2020 года это реализовано веб-браузерами Node, Chrome, Edge и Safari, но не Firefox.) Такие классы символов основаны на характеристиках, определенных стандартом Unicode, и представляемый ими набор символов с развитием Unicode может измениться.

Класс символов `\d` соответствует только цифрам ASCII. Если вы хотите соответствия с одной десятичной цифре из любой мировой системы записи, тогда можете применять `/\p{Decimal_Number}/u`. При желании соответствовать любому символу, который *не* является десятичной цифрой в любом языке, можете сделать `p` заглавной и написать `\P{Decimal_Number}`. Если вы хотите соответствия с любым похожим на число символом, включая дроби и римские цифры, то можете использовать `\p{Number}`. Обратите внимание, что `Decimal_Number` и `Number` не специфичны для JavaScript или для грамматики регулярных выражений: они представляют собой названия категорий символов, определенных стандартом Unicode.

Класс символов `\w` работает только с текстом ASCII, но с помощью `\p` мы можем приблизиться к интернационализированной версии наподобие:

```
/[\p{Alphabetic}\p{Decimal_Number}\p{Mark}]/u
```

(Хотя для полной совместимости со сложностью мировых языков на самом деле нам необходимо добавить категории `Connector_Punctuation` и `Join_Control`.)

В качестве финального примера синтаксис `\p` также позволяет определять регулярные выражения, которые соответствуют символам из специфического алфавита или системы письма:

```
let greekLetter = /\p{Script=Greek}/u;
let cyrillicLetter = /\p{Script=Cyrillic}/u;
```

Повторение

С помощью изученного до сих пор синтаксиса регулярных выражений вы можете описать число из двух цифр как `/\d\d/` и число из четырех цифр как `/\d\d\d\d/`. Но нет никакого способа описать, скажем, число, которое имеет любое количество цифр, или строку из трех букв с последующей необязательной цифрой. В таких более сложных шаблонах используется синтаксис регулярных выражений, который указывает, сколько раз может повторяться какой-то элемент регулярного выражения.

Символы, задающие повторение, всегда находятся после шаблона, к которому они применяются. Поскольку определенные типы повторения настолько распространены, существуют специальные символы для представления этих случаев. Например, `+` соответствует одному и более вхождений предшествующего шаблона. Синтаксис повторения подытожен в табл. 11.4.

Таблица 11.4. Символы повторения в регулярных выражениях

Символ	Что означает
{*n*,*m*}	Соответствует предшествующему элементу, по крайней мере, *n* раз, но не более *m* раз
{*n*,}	Соответствует предшествующему элементу *n* или больше раз
{*n*}	Соответствует в точности *n* вхождениям предшествующего элемента
?	Соответствует нулю или одному вхождению предшествующего элемента. То есть предшествующий элемент необязателен. Эквивалентно `{0,1}`
+	Соответствует одному или большему количеству вхождений предшествующего элемента. Эквивалентно `{1,}`
*	Соответствует нулю или большему количеству вхождений предшествующего элемента. Эквивалентно `{0,}`

Ниже приведены примеры:

```
let r = /\d{2,4}/; // Соответствует цифрам в количестве
                   // от двух до четырех
r = /\w{3}\d?/;    // Соответствует в точности трем символам слов
                   // и необязательной цифре
r = /\s+java\s+/;  // Соответствует "java" с одним
                   // и более пробелов до и после
r = /[^(]*/;   // Соответствует нулю или большему количеству символов,
               // которые не являются открывающими круглыми скобками
```

Обратите внимание, что во всех примерах спецификаторы повторения применяются к одиночному символу или классу символов, который предваряет их. При желании сопоставлять с более сложными выражениями вам понадобится определить группу в круглых скобках, как объясняется далее в главе.

Будьте осторожны, когда используете символы повторения * и ?. Так как они могут соответствовать нулевому количеству вхождений того, что их предваряет, то им разрешено ничему не соответствовать. Скажем, регулярное выражение `/a*/` на самом деле соответствует строке `"bbbb"`, потому что строка содержит ноль вхождений буквы a!

Нежадное повторение

Символы повторения из табл. 11.4 дают соответствие максимально возможное количество раз, пока обеспечивается сопоставление для любых последующих частей регулярного выражения. Мы называем такое повторение "жадным". Также можно указать, что повторение должно выполняться нежадным способом. Просто добавьте к символу или символам повторения вопросительный знак: `??`, `+?`, `*?` или даже `{1,5}?`. Например, регулярное выражение `/a+/` соответствует одному или большему количеству вхождений буквы a. Когда применяется к строке `"aaa"`, оно соответствует всем трем буквам. Но `/a+?/` соответствует одному или большему количеству вхождений буквы a, выполняя сопоставление только с необходимым числом символов. В случае применения к той же строке `"aaa"` такой шаблон соответствует лишь первой букве a.

Использование нежадного повторения не всегда приводит к ожидаемым результатам. Рассмотрим шаблон `/a+b/`, соответствующий одной или большему количеству буквы a, за которой следует буква b. Когда он применяется к строке `"aaab"`, то соответствует целой строке. А теперь давайте воспользуемся нежадной версией: `/a+?b/`. Этот шаблон должен соответствовать букве b, которой предшествует наименьшее из возможных количество буквы a. В случае применения к той же самой строке `"aaab"` вы могли бы ожидать соответствия только одной букве a и последней букве b. Однако в действительности шаблон `/a+?b/` соответствует целой строке, как и его жадная версия. Причина в том, что сопоставление с образцом в виде регулярного выражения выполняется путем нахождения первой позиции в строке, где возможно соответствие. Поскольку соответствие возможно, начиная с первого символа строки, то менее длинные соответствия, которые начинаются с последующих символов, даже не принимаются во внимание.

Чередование, группирование и ссылки

Грамматика регулярных выражений включает специальные символы для указания альтернатив, группирования подвыражений и ссылки на предшествующие подвыражения. Символ `|` отделяет альтернативы друг от друга. Скажем, `/ab|cd|ef/` соответствует строке `"ab"` или строке `"cd"` или строке `"ef"`. А шаблон `/\d{3}|[a-z]{4}/` соответствует либо трем цифрам, либо четырем буквам нижнего регистра.

Имейте в виду, что альтернативы просматриваются слева направо до тех пор, пока не обнаруживается совпадение. Если альтернатива слева дает совпадение, то альтернатива справа игнорируется, даже когда она обеспечила бы "лучшее" совпадение. Таким образом, шаблон `/a|ab/`, примененный к строке `"ab"`, соответствует только первой букве.

Круглые скобки имеют несколько целевых назначений в регулярных выражениях. Одно из них — группирование отдельных элементов в одиночное подвыражение, чтобы элементы можно было трактовать как единое целое посредством `|`, `*`, `+`, `?` и т.д. Например, `/java(script)?/` соответствует строке `"java"`, за которой следует необязательная строка `"script"`, а `/(ab|cd)+|ef/` соответствует либо строке `"ef"`, либо одному или большему количеству повторений строки `"ab"` или `"cd"`.

Еще одно целевое назначение круглых скобок в регулярных выражениях — определение подшаблонов внутри полного шаблона. Когда регулярное выражение успешно сопоставляется с целевой строкой, то можно извлекать порции целевой строки, которые соответствуют любому отдельному подшаблону в круглых скобках. (Позже в этом разделе вы увидите, как получать такие совпадающие подстроки.) Скажем, пусть вы ищете одну или несколько букв нижнего регистра, за которыми следует одна или больше цифр. Вы могли бы использовать шаблон `/[a-z]+\d+/`. Но предположим, что вас в действительности интересуют только цифры в конце каждого совпадения. Если вы поместите эту часть шаблона в круглые скобки, т.е. `(/[a-z]+(\d+)/)`, то сможете извлекать цифры из любых найденных совпадений, как будет объясняться далее.

Подвыражения в круглых скобках также позволяют ссылаться обратно на подвыражение позже в том же самом регулярном выражении, указывая после символа `\` цифру или цифры. Цифры относятся к позиции подвыражения в круглых скобках внутри регулярного выражения. Например, `\1` ссылается обратно на первое подвыражение, а `\3` — на третье. Обратите внимание, что поскольку подвыражения могут быть вложенными в другие подвыражения, в расчет принимается только позиции левых круглых скобок. Скажем, в показанном ниже регулярном выражении на вложенное подвыражение `([Ss]cript)` ссылаются с помощью `\2`:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

Ссылка на предыдущее подвыражение регулярного выражения относится *не* к шаблону для этого подвыражения, а к тексту, который соответствует шаблону. Таким образом, ссылки можно применять для навязывания ограничения, заключающегося в том, что отдельные порции строки должны содержать в точности те же самые символы. Например, приведенное далее регулярное выражение соответствует нулю или большему количеству символов в одинарных или двойных кавычках. Тем не менее, оно не требует совпадения открывающих и закрывающих кавычек (т.е. чтобы обе кавычки были одинарными или двойными):

```
/['"][^'"]*['"]/
```

Совпадения кавычек можно потребовать с использованием ссылки:

```
/(['"])[^'"]*\1/
```

Здесь \1 соответствует всему тому, с чем совпадает первое подвыражение в круглых скобках. В данном примере принудительно применяется ограничение, касающееся того, что закрывающая кавычка обязана совпадать с открывающей кавычкой. Регулярное выражение не разрешает использовать одинарные кавычки внутри строк в двойных кавычках и наоборот. (Применять ссылку внутри класса символов незаконно, а потому нельзя записать так: /(['"])[^\1]*\1/.)

При дальнейшем раскрытии API-интерфейса RegExp вы увидите, что такой способ ссылки на подвыражение в круглых скобках является мощной особенностью операций поиска и замены, основанных на регулярных выражениях.

Группировать элементы в регулярном выражении также можно, не создавая числовую ссылку на них. Вместо группирования элементов внутри (и) начните группу с (?: и закончите на). Взгляните на следующий шаблон:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

В этом примере подвыражение (?:[Ss]cript) используется просто для группирования, чтобы к группе можно было применить символ повторения ?. Такие модифицированные круглые скобки не производят ссылку, так что в данном регулярном выражении \2 ссылается на текст, соответствующий (fun\w*).

В табл. 11.5 перечислены операции чередования, группирования и ссылки в регулярных выражениях.

Таблица 11.5. Символы чередования, группирования и ссылки в регулярных выражениях

Символ	Что означает
|	Чередование: соответствует либо подвыражению слева, либо подвыражению справа
(...)	Группирование: группирует элементы в единое целое, которое может использоваться с *, +, ?, | и т.д. Также запоминает символы, которые соответствуют этой группе, для применения в последующих ссылках
(?:...)	Только группирование: группирует элементы в единое целое, но не запоминает символы, которые соответствуют этой группе
\n	Соответствует тем же символам, которые дали совпадение, когда впервые сопоставлялась группа номер *n*. Группы — это подвыражения внутри (возможно вложенных) круглых скобок. Номера назначаются группам путем подсчета левых круглых скобок слева направо. Группы, сформированные с помощью (?:, не нумеруются

Именованные захватывающие группы

В ES2018 стандартизируется новое средство, которое способно делать регулярные выражения более самодокументированными и легкими для понимания. Новое средство называется "именованными захватывающими группами" (named capturing group) и дает возможность ассоциировать имя с каждой левой круглой скобкой в регулярном выражении, чтобы можно было ссылаться на совпадающий текст по имени, а не по номеру. Не менее

важно и то, что использование имен позволяет тому, кто читает код, лучше понимать назначение той или иной порции регулярного выражения. По состоянию на начало 2020 года средство реализовано в веб-браузерах Node, Chrome, Edge и Safari, но не в Firefox.

Для именования группы применяйте `(?<...>` вместо `(` и укажите имя между угловыми скобками. Скажем, следующее регулярное выражение можно использовать для проверки формата последней строки почтового адреса в США:

```
/(?<city>\w+) (?<state>[A-Z]{2}) (?<zipcode>\d{5})(?<zip9>-\d{4})?/
```

Обратите внимание, сколько контекста обеспечивают имена групп, чтобы облегчить понимание регулярного выражения. Когда мы будем обсуждать методы `replace()` и `match()` класса `String` и метод `exec()` класса `RegExp` в подразделе 11.3.2, вы увидите, каким образом API-интерфейс `RegExp` позволяет ссылаться на текст, соответствующий каждой из этих групп, по имени, а не по позиции.

Если вас интересует обратная ссылка на именованную захватывающую группу внутри регулярного выражения, то вы можете делать ее также по имени. В предыдущем примере мы имели возможность применять "обратную ссылку" регулярного выражения, чтобы создать объект `RegExp`, который соответствовал бы строке в одинарных или двойных кавычках, где открывающая и закрывающая кавычки обязаны совпадать. Вот как можно переписать `RegExp` с использованием именованной захватывающей группы и именованной обратной ссылки:

```
/(?<quote>['"])[^'"]*\k<quote>/
```

Здесь `\k<quote>` — именованная обратная ссылка на именованную группу, которая захватывает открывающую кавычку.

Указание позиции соответствия

Как было описано ранее, многие элементы регулярного выражения соответствуют одиночному символу в строке. Например, `\s` соответствует одиночному пробельному символу. Другие элементы регулярного выражения соответствуют позициям между символами, а не фактическим символам. Скажем, `\b` соответствует границе слова ASCII — границе между `\w` (символ слова ASCII) и `\W` (символ не слова) или границе между символом слова ASCII и началом или концом строки[4]. Элементы вроде `\b` не задают никаких символов, подлежащих применению в совпавшей строке; однако, они указывают допустимые позиции, в которых совпадение может произойти. Иногда такие элементы называют *якорями регулярных выражений*, потому что они прикрепляют шаблон к специфической позиции в строке поиска. Наиболее часто используемыми якорными элементами являются `^`, который привязывает шаблон к началу строки, и `$`, прикрепляющий шаблон к концу строки.

Например, чтобы найти слово "JavaScript" в отдельной строке, вы можете применить регулярное выражение `/^JavaScript$/`. Если вы хотите искать

[4] Исключая класс символов (в квадратных скобках), где `\b` соответствует символу забоя.

"Java" как отдельное слово (не как префикс в "JavaScript"), тогда можете опробовать шаблон /\sJava\s/, который требует наличия пробела до и после слова. Но этому решению присущи две проблемы. Во-первых, оно не обеспечит совпадение со словом "Java" в начале или в конце строки, а только если слово появляется с пробелами с обеих сторон. Во-вторых, когда этот шаблон находит соответствие, то возвращаемая им совпавшая строка содержит головной и хвостовой пробелы, т.е. не совсем то, что нужно. Таким образом, вместо сопоставления с фактическими символами пробелов посредством \s необходимо сопоставлять с границами слова (или привязываться к ним) с помощью \b. Результирующее выражение выглядит как /\bJava\b/. Элемент \B прикрепляет соответствие к позиции, которая не является границей слова. Следовательно, шаблон /\B[Ss]cript/ соответствует "JavaScript" и "postscript", но не "script" или "Scripting".

Вы также можете использовать в качестве якорных условий произвольные регулярные выражения. Поместив выражение внутрь символов (?= и), вы получите утверждение просмотра вперед, которое указывает, что указанные символы должны совпадать, но не включаться в найденное соответствие. Например, чтобы получить совпадение с названием распространенного языка программирования, но только если за ним следует двоеточие, вы могли бы применить шаблон /[Jj]ava([Ss]cript)?(?=\:)/. Этот шаблон соответствует слову "JavaScript" в "JavaScript: The Definitive Guide", но не соответствует слову "Java" в "Java in a Nutshell", т.к. за ним нет двоеточия.

Если взамен вы введете утверждение с (?!, то получите утверждение отрицательного просмотра вперед, которое указывает, что следующие символы не должны давать совпадение. Скажем, шаблон /Java(?!Script)([A-Z]\w*)/ соответствует слову "Java", за которым идет заглавная буква и любое количество дополнительных символов слов ASCII при условии, что за словом "Java" не находится "Script". Шаблон соответствует "JavaBeans", но не "Javanese", а также "JavaScrip", но не "JavaScript" или "JavaScripter". Якорные символы в регулярных выражениях перечислены в табл. 11.6.

Таблица 11.6. Якорные символы в регулярных выражениях

Символ	Что означает
^	Соответствует началу строки или при наличии флага m началу линии в строке
$	Соответствует концу строки или при наличии флага m концу линии в строке
\b	Соответствует границе слова, т.е. позиции между символом \w и символом \W либо между символом \w и началом или концом строки. (Тем не менее, имейте в виду, что [\b] соответствует символу забоя.)
\B	Соответствует позиции не в границе слова
(?=*p*)	Утверждение положительного просмотра вперед. Требует, чтобы последующие символы соответствовали шаблону *p*, но не включает их в найденное соответствие
(?!*p*)	Утверждение отрицательного просмотра вперед. Требует, чтобы последующие символы не соответствовали шаблону *p*

Утверждения просмотра назад

В ES2018 синтаксис регулярных выражений был расширен, чтобы сделать возможными утверждения "просмотра назад". Они похожи на утверждения просмотра вперед, но обращаются к тексту перед текущей позицией совпадения. По состоянию на начало 2020 года утверждения просмотра назад реализованы в веб-браузерах Node, Chrome и Edge, но не в Firefox или Safari.

Указывайте утверждение положительного просмотра назад с помощью `(?<=...)` и утверждение отрицательного просмотра назад посредством `(?<!...)`. Например, если вы работали с почтовыми адресами в США, то могли бы выполнять сопоставление с почтовым индексом из пяти цифр, но только когда он следует за двухбуквенным сокращением названия штата:

```
/(?<= [A-Z]{2} )\d{5}/
```

И с помощью утверждения отрицательного просмотра назад вы можете выполнять сопоставление со строкой цифр, которой не предшествует символ валюты:

```
/(?<![\p{Currency_Symbol}\d.])\d+(\.\d+)?/u
```

Флаги

Каждое регулярное выражение может иметь один и более ассоциированных с ним флагов, изменяющих его поведение сопоставления. В JavaScript определены шесть возможных флагов, каждый из которых представлен одиночной буквой. Флаги указываются после второго символа / литерала регулярного выражения или в виде строки, передаваемой во втором аргументе конструктору `RegExp()`. Ниже описаны поддерживаемые флаги.

- **`g`**. Флаг `g` указывает, что регулярное выражение является "глобальным", т.е. мы намерены его использовать для нахождения всех совпадений внутри строки, а только первого совпадения. Флаг `g` не изменяет способ сопоставления с образцом, но, как будет показано позже, он изменяет поведение метода `match()` класса `String` и метода `exec()` класса `RegExp` во многих важных аспектах.
- **`i`**. Флаг `i` указывает, что сопоставление с образцом должно быть нечувствительным к регистру.
- **`m`**. Флаг `m` указывает, что сопоставление с образцом должно выполняться в "многострочном" режиме. Он говорит о том, что объект `RegExp` будет применяться с многострочными строками, а якоря `^` и `$` должны соответствовать началу и концу строки плюс началу и концу индивидуальных линий внутри строки.
- **`s`**. Подобно `m` флаг `s` также полезен при работе с текстом, который включает символы новой строки. Обычно точка (`.`) в регулярном выражении соответствует любому символу кроме разделителя строк. Однако когда используется флаг `s`, точка будет соответствовать любому символу, в том числе разделителям строк. Флаг `s` был добавлен к JavaScript в ES2018 и по

состоянию на начало 2020 года он поддерживается в Node, Chrome, Edge и Safari, но не в Firefox.

- **u**. Флаг u означает Unicode и заставляет регулярное выражение соответствовать полным кодовым точкам Unicode, а не 16-битным значениям. Флаг u был введен в ES6, и вы должны выработать привычку применять его во всех регулярных выражениях, если только не существует веская причина не поступать так. Если вы не используете этот флаг, тогда ваши объекты RegExp не будут нормально работать с текстом, содержащим эмотиконы и другие символы (включая многие китайские иероглифы), которые требуют более 16 бит. В отсутствие флага u символ . соответствует любому одному 16-битному значению UTF-16. Тем не менее, при наличии флага u символ . соответствует одной кодовой точке Unicode, в том числе содержащей свыше 16 бит. Установка флага u в объекте RegExp также позволяет применять новую управляющую последовательность `\u{...}` для символа Unicode и делает возможной запись `\p{...}` для классов символов Unicode.
- **y**. Флаг y указывает на то, что регулярное выражение является "липким" и должно совпадать в начале строки или на первом символе, который следует за предыдущим соответствием. При использовании с регулярным выражением, предназначенным для поиска одиночного совпадения, флаг y фактически трактует это регулярное выражение так, как если бы оно начиналось с ^ с целью его прикрепления к началу строки. Этот флаг более полезен с регулярными выражениями, которые применяются многократно для поиска всех совпадений внутри строки. В таком случае он инициирует специальное поведение метода `match()` класса `String` и метода `exec()` класса RegExp, чтобы обеспечить прикрепление каждого последующего совпадения к позиции в строке, в которой закончилось последнее совпадение.

Описанные выше флаги могут указываться в любой комбинации и в любом порядке. Например, если вы хотите, чтобы ваше регулярное выражение распознавало Unicode для выполнения сопоставления, нечувствительного к регистру, и намереваетесь использовать его для поиска множества совпадений внутри строки, то должны указать флаги `uig`, `gui` или любую другую перестановку этих трех букв.

11.3.2. Строковые методы для сопоставления с образцом

До сих пор мы описывали грамматику, применяемую для определения регулярных выражений, но не объясняли, как такие регулярные выражения можно практически использовать в коде JavaScript. Теперь мы переходим к рассмотрению API-интерфейса для работы с объектами RegExp. Текущий подраздел начинается с обсуждения строковых методов, которые применяют регулярные выражения для выполнения сопоставления с образцом и операций поиска и замены. В последующих подразделах обсуждение сопоставления с образцом с помощью регулярных выражений JavaScript продолжится исследованием класса RegExp и его методов и свойств.

search()

Строки поддерживают четыре метода, которые используют регулярные выражения. Метод search() — самый простой из них. Он принимает аргумент с регулярным выражением и возвращает либо символьную позицию начала первой совпадающей подстроки, либо -1, если совпадения не обнаружены:

```
"JavaScript".search(/script/ui)     // => 4
"Python".search(/script/ui)         // => -1
```

Если аргумент метода search() не является регулярным выражением, тогда он сначала преобразуется в него путем передачи конструктору RegExp. Метод search() не поддерживает глобальный поиск; он игнорирует флаг g в своем аргументе с регулярным выражением.

replace()

Метод replace() выполняет операцию поиска и замены. Он принимает регулярное выражение в своем первом аргументе и строку замены во втором. Метод replace() ищет в строке, на которой он вызван, совпадения с указанным шаблоном. Если в регулярном выражении указан флаг g, тогда метод replace() заменяет все совпадения в строке строкой замены; в противном случае он заменяет только первое найденное совпадение. Если в первом аргументе replace() передается строка, а не регулярное выражение, то метод ищет эту строку буквально, не преобразуя ее в регулярное выражение посредством конструктора RegExp(), как поступает search(). В качестве примера вот как можно применить replace() для обеспечения единообразного использования заглавных букв в слове "JavaScript" во всей текстовой строке:

```
// Независимо от того, как употребляются заглавные буквы,
// заменить строку корректным вариантом.
text.replace(/javascript/gi, "JavaScript");
```

Однако метод replace() обладает большей мощностью, чем здесь продемонстрировано. Вспомните, что подвыражения в круглых скобках внутри регулярного выражения нумеруются слева направо, а регулярное выражение запоминает текст, с которым дает совпадение каждое подвыражение. Если в строке замены присутствует символ $ с последующей цифрой, тогда replace() заменяет эти два символа текстом, который соответствует указанному подвыражению. Такая возможность очень удобна. Скажем, вы можете ее применять для замены кавычек в строке другими символами:

```
// Цитата выглядит как кавычка, за которой следует любое количество
// символов, отличающихся от кавычек (их мы захватываем), и затем
// еще одна кавычка.
let quote = /"([^"]*)"/g;

// Заменить прямые кавычки типографскими, оставив
// прежним цитируемый текст (хранящийся в $1).
'He said "stop"'.replace(quote, '«$1»')  // => 'He said «stop»'
```

Если в объекте RegExp используются именованные захватывающие группы, то вы можете ссылаться на совпадающий текст по имени, а не по номеру:

```
let quote = /"(?<quotedText>[^"]*)"/g;
'He said "stop"'.replace(quote, '«$<quotedText>»') // => 'He said «stop»'
```

Вместо передачи строки замены во втором аргументе `replace()` вы также можете передать функцию, которая будет вызываться для вычисления значения замены. Функция замены вызывается с рядом аргументов. Первый аргумент — полный совпадающий текст. Если объект RegExp имеет захватывающие группы, тогда в качестве аргументов передаются подстроки, захваченные этими группами. Следующий аргумент — позиция внутри строки, в которой было найдено совпадение. Затем передается полная строка, на которой был вызван метод `replace()`. Наконец, если объект RegExp содержит именованные захватывающие группы, то последним аргументом функции замены будет объект, свойствами которого являются имена захватывающих групп, а значениями — совпадающий текст. Например, ниже показан код, в котором функция замены применяется для преобразования десятичных целых чисел в строке в шестнадцатеричные:

```
let s = "15 times 15 is 225";
s.replace(/\d+/gu, n => parseInt(n).toString(16))  // => "f times f is e1"
```

match()

Метод `match()` — наиболее универсальный из методов работы с регулярными выражениями в классе `String`. Он принимает в своем единственном аргументе регулярное выражение (или преобразует значение аргумента в регулярное выражение, передавая его конструктору `RegExp()`) и возвращает массив, который содержит результаты сопоставления или `null`, если совпадения не найдены. Если в регулярном выражении установлен флаг `g`, тогда метод возвращает массив всех совпадений, обнаруженных в строке. Вот пример:

```
"7 plus 8 equals 15".match(/\d+/g)                    // => ["7", "8", "15"]
```

Если в регулярном выражении флаг `g` не установлен, то метод `match()` не выполняет глобальный поиск; он просто ищет первое совпадение. В неглобальном случае метод `match()` по-прежнему возвращает массив, но элементы массива будут совершенно другими. Без флага `g` первый элемент возвращаемого массива представляет собой совпадающую строку, а оставшиеся элементы — подстроки, которые соответствуют захватывающим группам в круглых скобках регулярного выражения. Таким образом, если `match()` возвращает массив `a`, то `a[0]` содержит полное совпадение, `a[1]` — подстроку, совпадающую с первым подвыражением в круглых скобках, и т.д. Чтобы провести параллель с методом `replace()`, элемент `a[1]` является той же строкой, что и `$1`, `a[2]` — то же самое, что и `$2`, и т.д.

Например, рассмотрим разбор URL[5] посредством следующего кода:

[5] Проводить разбор URL с помощью регулярных выражений — не лучшая идея. В разделе 11.9 представлен более надежный анализатор URL.

```
// Очень простое регулярное выражение для разбора URL
let url = /(\w+):\/\/([\w.]+)\/(\S*)/;
let text = "Visit my blog at http://www.example.com/~david";
let match = text.match(url);
let fullurl, protocol, host, path;
if (match !== null) {
   fullurl = match[0];  // fullurl == "http://www.example.com/~david"
   protocol = match[1]; // protocol == "http"
   host = match[2];     // host == "www.example.com"
   path = match[3];     // path == "~david"
}
```

В таком неглобальном случае массив, возвращаемый `match()`, помимо нумерованных элементов также имеет ряд объектных свойств. Свойство `input` ссылается на строку, на которой метод `match()` вызывался. Свойство `index` хранит позицию внутри строки, с которой начинается совпадение. Если же регулярное выражение содержит именованные захватывающие группы, тогда возвращаемый массив имеет и свойство `groups`, значением которого будет объект. Свойства этого объекта соответствуют именам захватывающих групп, а значения — совпадающему тексту. Мы можем переписать предыдущий пример разбора URL, как показано ниже:

```
let url = /(?<protocol>\w+):\/\/(?<host>[\w.]+)\/(?<path>\S*)/;
let text = "Visit my blog at http://www.example.com/~david";
let match = text.match(url);
match[0]                // => "http://www.example.com/~david"
match.input             // => text
match.index             // => 17
match.groups.protocol   // => "http"
match.groups.host       // => "www.example.com"
match.groups.path       // => "~david"
```

Вы видели, что метод `match()` ведет себя по-разному в зависимости от того, установлен ли флаг `g` в `RegExp`. Также имеются важные, но менее существенные отличия в поведении, когда установлен флаг `y`. Вспомните, что флаг `y` делает регулярное выражение "липким", ограничивая то, где в строке могут начинаться совпадения. Если в объекте `RegExp` установлены оба флага, `g` и `y`, тогда метод `match()` возвращает массив совпадающих строк, как в случае, когда установлен флаг `g` без флага `y`. Но первое совпадение обязано стартовать с начала строки, а каждое последующее совпадение должно начинаться с символа, следующего сразу за предыдущим совпадением.

Если флаг `y` установлен без флага `g`, то метод `match()` пытается найти одиночное совпадение, которое по умолчанию ограничено началом строки. Тем не менее, вы можете изменить такую стандартную стартовую позицию, установив свойство `lastIndex` объекта `RegExp` в индекс, с которого хотите начинать сопоставление. Если совпадение найдено, тогда свойство `lastIndex` автоматически обновится позицией первого символа после совпадения, поэтому в случае повторного вызова метода `match()` он будет искать последующее совпадение. (`lastIndex` (индекс последнего сопоставления) может выглядеть странным именем для свойства,

указывающего позицию, с которой должно начинаться *следующее* (next) сопоставление. Мы встретимся с ним снова, когда будем исследовать метод `exec()` класса `RegExp`, и его имя в том контексте может иметь больше смысла.)

```
let vowel = /[aeiou]/y; // "Липкое" выражение сопоставления с гласными
"test".match(vowel)     // => null: "test" не начинается с гласной
vowel.lastIndex = 1;    // Указать другую позицию для сопоставления
"test".match(vowel)[0] // => "e": найдена гласная в позиции 1
vowel.lastIndex   // => 2: свойство lastIndex автоматически обновилось
"test".match(vowel)     // => null: нет гласных в позиции 2
vowel.lastIndex         // => 0: после неудавшегося поиска свойство
                        //       lastIndex переустанавливается
```

Нелишне отметить, что передача неглобального регулярного выражения методу `match()` строки — это то же самое, что и передача строки методу `exec()` регулярного выражения: в обоих случаях возвращаемый массив и его свойства будут одинаковыми.

`matchAll()`

Метод `matchAll()` определен в ES2020 и по состоянию на начало 2020 года реализован современными веб-браузерами и Node. Метод `matchAll()` ожидает объект `RegExp` с установленным флагом `g`. Однако вместо возвращения массива совпадающих подстрок, как делает `match()`, он возвращает итератор, выдающий объекты совпадения такого вида, которые метод `match()` возвращает, когда используется с неглобальным `RegExp`. Таким образом, метод `matchAll()` предлагает самый легкий и универсальный способ прохода по всем совпадениям внутри строки. Вот как можно применять `matchAll()` для прохода в цикле по словам в строке текста:

```
// Один или большее количество алфавитных символов Unicode
// между границами слов
const words = /\b\p{Alphabetic}+\b/gu;  // \p пока не поддерживается
                                        // в Firefox
const text = "This is a nanve test of the matchAll() method.";
for(let word of text.matchAll(words)) {
   console.log(`Found '${word[0]}' at index ${word.index}.`);
}
```

Вы можете устанавливать свойство `lastIndex` объекта `RegExp`, чтобы сообщить методу `matchAll()`, с какого индекса в строке начинать сопоставление. Тем не менее, в отличие от других методов сопоставления с образцом метод `matchAll()` никогда не модифицирует свойство `lastIndex` объекта `RegExp`, на котором вы его вызываете, что снижает вероятность появления ошибок в вашем коде.

`split()`

Последним методом класса `String` для работы с регулярными выражениями является `split()`. Он разбивает строку, на которой вызывается, на массив подстрок, используя аргумент в качестве разделителя. Метод `split()` может применяться со строковым аргументом, например:

```
"123,456,789".split(",")          // => ["123", "456", "789"]
```

Метод `split()` также может принимать в своем аргументе регулярное выражение, что позволяет указывать более общие разделители. Ниже мы вызываем его с разделителем, который включает произвольное количество пробельных символов с обеих сторон:

```
"1, 2, 3,\n4, 5".split(/\s*,\s*/)  // => ["1", "2", "3", "4", "5"]
```

Удивительно, но если вы вызовите `split()` с разделителем RegExp и регулярное выражение содержит захватывающие группы, то текст, который соответствует захватывающим группам, будет включен в возвращаемый массив. Например:

```
const htmlTag = /<([^>]+)>/;  // Символ <, за которым следует один или
                              // больше символов, отличающихся от >, и затем символ >
"Testing<br/>1,2,3".split(htmlTag)  // => ["Testing", "br/", "1,2,3"]
```

11.3.3. Класс RegExp

В этом подразделе будут описаны конструктор `RegExp()`, свойства экземпляров RegExp и два важных метода сопоставления с образцом, определенные в классе RegExp.

Конструктор `RegExp()` принимает один или два строковых аргумента и создает новый объект RegExp. Первый аргумент конструктора представляет собой строку, содержащую тело регулярного выражения — текст, который находился бы между символами обратной косой черты в литерале регулярного выражения. Обратите внимание, что строковые литералы и регулярные выражения используют символ `\` для управляющих последовательностей, поэтому при передаче регулярного выражения конструктору `RegExp()` как строкового литерала вы обязаны заменить каждый символ `\` парой `\\`. Второй аргумент `RegExp()` необязателен. Если он предоставлен, то задает флаги регулярного выражения, которые должны быть `g`, `i`, `m`, `s`, `u`, `y` или любой комбинацией указанных букв.

Вот пример:

```
// Найти все числа из пяти цифр в строке. Обратите внимание
// на дублирование символа \, т.е. \\.
let zipcode = new RegExp("\\d{5}", "g");
```

Конструктор `RegExp()` полезен, когда регулярное выражение создается динамически и потому не может быть представлено с помощью синтаксиса литералов регулярных выражений. Скажем, для поиска строки, введенной пользователем, регулярное выражение должно создаваться во время выполнения посредством `RegExp()`.

Вместо передачи строки в первом аргументе `RegExp()` вы также можете передавать объект RegExp, что позволяет копировать регулярное выражение и изменять его флаги:

```
let exactMatch = /JavaScript/;
let caseInsensitive = new RegExp(exactMatch, "i");
```

Свойства RegExp

Объекты RegExp имеют перечисленные далее свойства.

- **source**. Это свойство, допускающее только чтение, представляет собой исходный текст регулярного выражения: символы, находящиеся между символами косой черты в литерале RegExp.
- **flags**. Это свойство, допускающее только чтение, является строкой, которая указывает набор букв, представляющих флаги для RegExp.
- **global**. Это булевское свойство, допускающее только чтение, которое равно true, если установлен флаг g.
- **ignoreCase**. Это булевское свойство, допускающее только чтение, которое равно true, если установлен флаг i.
- **multiline**. Это булевское свойство, допускающее только чтение, которое равно true, если установлен флаг m.
- **dotAll**. Это булевское свойство, допускающее только чтение, которое равно true, если установлен флаг s.
- **unicode**. Это булевское свойство, допускающее только чтение, которое равно true, если установлен флаг u.
- **sticky**. Это булевское свойство, допускающее только чтение, которое равно true, если установлен флаг которое равно true, если установлен флаг y.
- **lastIndex**. Это свойство представляет собой целое число, допускающее чтение/запись. Для шаблонов с флагами g или y оно указывает символьную позицию, с которой должен начинаться следующий поиск. Применяется методами exec() и test(), описанными в следующих двух подразделах.

test()

Метод test() класса RegExp предлагает простейший способ использования регулярного выражения. Он принимает единственный строковый аргумент и возвращает true, если строка соответствует шаблону, и false в противном случае.

Метод test() просто вызывает (гораздо более сложный) метод exec(), описанный в следующем подразделе, и возвращает true, если exec() возвращает значение не null. По указанной причине, если вы применяете метод test() с объектом RegExp, который использует флаг g или y, тогда его поведение зависит от значения свойства lastIndex объекта RegExp, что может неожиданно измениться. Дополнительные сведения приведены во врезке "Свойство lastIndex и многократное использование RegExp" далее в главе.

exec()

Метод exec() класса RegExp предлагает наиболее универсальный и мощный способ работы с регулярными выражениями. Он принимает единственный строковый аргумент и ищет совпадение в этой строке. Если совпадение не най-

дено, то метод возвращает null. Однако если совпадение найдено, тогда exec() возвращает массив, похожий на массив, возвращаемый методом match() для неглобального поиска. Элемент 0 массива содержит строку, которая соответствует регулярному выражению, а последующие элементы массива хранят подстроки, соответствующие любым захватывающим группам. Возвращаемый массив также имеет именованные свойства: свойство index содержит символьную позицию, где обнаружено совпадение, свойство input указывает искомую строку, а свойство groups, если определено, ссылается на объект, который хранит подстроки, соответствующие любым именованным захватывающим группам.

В отличие от метода match() класса String метод exec() возвращает массив одного и того же вида независимо от наличия флага g в регулярном выражении. Вспомните, что match() возвращает массив совпадений, когда ему передается глобальное регулярное выражение. Напротив, exec() всегда возвращает единственное совпадение и предоставляет полную информацию об этом совпадении. В случае вызова exec() на регулярном выражении, в котором установлен либо флаг g, либо флаг y, он обращается к свойству lastIndex объекта RegExp, чтобы выяснить, где начинать поиск совпадения. (И если флаг y установлен, то он также ограничивает совпадение началом в данной позиции.) Для вновь созданного объекта RegExp свойство lastIndex равно 0 и поиск стартует с начала строки. Но каждый раз, когда метод exec() успешно находит совпадение, он обновляет свойство lastIndex индексом символа, расположенного сразу после совпавшего текста. Если метод exec() терпит неудачу с поиском совпадения, то он переустанавливает lastIndex в 0. Такое особое поведение позволяет многократно вызывать exec() для прохода по всем совпадениям с регулярным выражением в строке. (Хотя, как уже было показано, в ES2020 и последующих версиях метод matchAll() класса String предлагает более легкий способ прохода по всем совпадениям.) Например, тело цикла в следующем коде будет выполняться дважды:

```
let pattern = /Java/g;
let text = "JavaScript > Java";
let match;
while((match = pattern.exec(text)) !== null) {
   console.log(`Matched ${match[0]} at ${match.index}`);
   console.log(`Next search begins at ${pattern.lastIndex}`);
}
```

Свойство lastIndex и многократное использование RegExp

Как вы уже видели, API-интерфейс регулярных выражений JavaScript отличается довольно высокой сложностью. Работа со свойством lastIndex при установленных флагах g и y является особенно неуклюжей частью этого API-интерфейса. Когда вы применяете упомянутые флаги, то должны проявлять крайнюю осторожность при вызове методов match(), exec() или test(), поскольку их поведение зависит от свойства lastIndex, а значение lastIndex зависит от того, что вы ранее делали с объектом RegExp. В итоге легко написать код, содержащий множество ошибок.

Предположим, например, что мы хотим отыскать индексы всех дескрипторов <p> внутри строки HTML-текста. Можно было бы написать код следующего вида:

```
let match, positions = [];
while((match = /<p>/g.exec(html)) !== null){//ВОЗМОЖЕН БЕСКОНЕЧНЫЙ ЦИКЛ
   positions.push(match.index);
}
```

Этот код не решает желаемую задачу. Если строка `html` содержит, по крайней мере, один дескриптор <p>, тогда цикл будет бесконечным. Проблема в том, что мы используем литерал `RegExp` в условии цикла `while`. На каждой итерации цикла мы создаем новый объект `RegExp` со свойством `lastIndex`, установленным в 0, поэтому метод `exec()` всегда начинает поиск с начала строки, и если есть совпадение, то он продолжит сопоставление снова и снова. Разумеется, решение заключается в однократном определении объекта `RegExp` и его сохранение в переменной, чтобы мы применяли тот же самый объект `RegExp` на каждой итерации цикла.

С другой стороны, временами многократное использование объекта `RegExp` — неправильный подход. Скажем, пусть мы хотим проходить в цикле по всем словам в словаре с целью нахождения слов, которые содержат пары удвоенных букв:

```
let dictionary = [ "apple", "book", "coffee" ];
let doubleLetterWords = [];
let doubleLetter = /(\w)\1/g;

for(let word of dictionary) {
   if (doubleLetter.test(word)) {
      doubleLetterWords.push(word);
   }
}
doubleLetterWords // => ["apple", "coffee"]: слово "book" пропущено!
```

Из-за того, что мы установили флаг `g` в объекте `RegExp`, свойство `lastIndex` изменяется после успешного нахождения совпадений, и метод `test()` (основанный на `exec()`) начинает поиск совпадения в позиции, указываемой `lastIndex`. После обнаружения "pp" в "apple" свойство `lastIndex` становится равным 3, а потому поиск в слове "book" начинается в позиции 3 и не заметит содержащегося в нем "oo".

Мы могли бы устранить проблему за счет удаления флага `g` (который на самом деле не нужен в этом конкретном примере), перемещения литерала `RegExp` внутрь тела цикла, чтобы он воссоздавался на каждой итерации, или явной переустановки свойства `lastIndex` в 0 перед каждым вызовом `test()`.

Мораль здесь в том, что `lastIndex` делает API-интерфейс `RegExp` подверженным ошибкам. Таким образом, будьте особенно внимательными, когда применяете флаг `g` или `y` и цикл. В ES2020 и последующих версиях используйте метод `matchAll()` класса `String` вместо `exec()`, чтобы обойти проблему, т.к. `matchAll()` не модифицирует `lastIndex`.

11.4. Дата и время

Класс Date является API-интерфейсом JavaScript для работы с датой и временем. Объект Date создается с помощью конструктора Date(). При вызове без аргументов он возвращает объект Date, который представляет текущую дату и время:

```
let now = new Date();      // Текущее время
```

Если вы передаете один числовой аргумент, то конструктор Date() интерпретирует его как количество миллисекунд, прошедших с 1 января 1970 года:

```
let epoch = new Date(0);  // Полночь, 1 января 1970 года,
                          // гринвичское среднее время
```

Если вы указываете два или больше целочисленных аргумента, то они интерпретируются как год, месяц, день недели, часы, минуты, секунды и миллисекунды в вашем локальном часовом поясе:

```
let century = new Date(2100,          // Год 2100
                       0,             // Январь
                       1,             // 1-е
                       2, 3, 4, 5);   // 02:03:04.005, локальное время
```

Одна индивидуальная особенность API-интерфейса Date заключается в том, что первый месяц года имеет номер 0, но первый день месяца — номер 1. Если вы опустите поля времени, то конструктор Date() по умолчанию выберет для них значения 0, устанавливая время в полночь.

Обратите внимание, что при вызове с множеством чисел конструктор Date() интерпретирует их с применением часового пояса, установленного на локальном компьютере. Если вы хотите указать дату и время в скоординированном всемирном времени (Universal Time Coordinated — UTC, оно же гринвичское среднее время, Greenwich Mean Time — GMT), тогда можете использовать Date.UTC(). Этот статический метод принимает те же аргументы, что и конструктор Date(), интерпретирует их в UTC и возвращает отметку времени в миллисекундах, которую вы можете передавать конструктору Date():

```
// Полночь в Англии, 1 января 2100 года
let century = new Date(Date.UTC(2100, 0, 1));
```

В случае вывода даты (скажем, посредством console.log(century)) по умолчанию она выводится в локальном часовом поясе. Если вы хотите отображать дату и время в UTC, тогда должны явно преобразовать ее в строку с помощью toUTCString() или toISOString().

Наконец, если вы передадите строку конструктору Date(), то он попытается разобрать ее как определение даты и времени. Конструктор способен разбирать даты, указанные в форматах, которые производят методы toString(), toUTCString() и toISOString():

```
let century = new Date("2100-01-01T00:00:00Z");  // Формат даты ISO
```

После создания объекта Date вам доступны разнообразные методы получения и установки, которые позволяют запрашивать и модифицировать поля года, месяца, дня недели, часов, минут, секунд и миллисекунд в объекте Date. Каждый такой метод имеет две формы: одна выполняет получение или установку с применением локального времени, а другая — с использованием времени UTC. Скажем, чтобы получить или установить год в объекте Date, вы должны применять метод getFullYear(), getUTCFullYear(), setFullYear() или setUTCFullYear():

```
let d = new Date();                         // Начать с текущей даты
d.setFullYear(d.getFullYear() + 1);         // Инкрементировать год
```

Для получения или установки других полей в Date замените "FullYear" в имени метода строками "Month", "Date", "Hours", "Minutes", "Seconds" или "Milliseconds". Некоторые методы установки даты дают возможность устанавливать сразу более одного поля. Методы setFullYear() и setUTCFullYear() также дополнительно позволяют устанавливать месяц и день месяца. А методы setHours() и setUTCHours() помимо поля часов разрешают устанавливать поля минут, секунд и миллисекунд.

Обратите внимание, что методы для запрашивания дня месяца называются getDate() и getUTCDate(). Более естественно именованные методы getDay() и getUTCDay() возвращают день недели (от 0 для воскресенья до 6 для субботы). Поле дня недели доступно только для чтения, поэтому соответствующего метода setDay() не предусмотрено.

11.4.1. Отметки времени

Внутренне даты JavaScript представлены в виде целых чисел, которые указывают количество миллисекунд, прошедших от полуночи 1 января 1970 года по времени UTC. Поддерживаются целые числа вплоть до 8 640 000 000 000 000, так что представления в миллисекундах JavaScript хватит для более 270 000 лет.

Для любого объекта Date метод getTime() возвращает упомянутое внутреннее значение, а метод setTime() устанавливает его. Таким образом, например, с помощью следующего кода можно прибавить 30 секунд к объекту Date:

```
d.setTime(d.getTime() + 30000);
```

Подобные миллисекундные значения иногда называют *отметками времени*, и часто работать с ними напрямую удобнее, чем с объектами Date. Статический метод Date.now() возвращает текущее время в виде отметки времени и полезен, когда нужно измерить время выполнения кода:

```
let startTime = Date.now();
reticulateSplines();  // Выполнить операцию, отнимающую много времени
let endTime = Date.now();
console.log(`Spline reticulation took ${endTime - startTime}ms.`);
```

Отметки времени с высоким разрешением

Отметки времени, возвращаемые методом `Date.now()`, измеряются в миллисекундах. В действительности миллисекунда для компьютера — относительно долгое время, и нередко желательно измерять истекшее время с более высокой точностью. Это позволяет делать функция `performance.now()`: она тоже возвращает отметку времени, основанную на миллисекундах, но возвращаемое значение — не целое число, а потому оно включает доли миллисекунды. Значение, возвращаемое `performance.now()`, не является абсолютной отметкой времени как значение `Date.now()`. Взамен оно просто указывает, сколько времени прошло с момента загрузки веб-страницы или с момента запуска процесса Node.

Объект `performance` входит в состав более крупного API-интерфейса Performance, который не определено стандартом ECMAScript, но реализован веб-браузерами и Node. Чтобы использовать объект `performance` в Node, его потребуется импортировать:

```
const { performance } = require("perf_hooks");
```

Разрешение высокоточного измерения времени в веб-сети может позволить недобросовестным веб-сайтам скрыто собирать сведения о посетителях, поэтому браузеры (особенно Firefox) могут по умолчанию понижать точность `performance.now()`. Как разработчик веб-приложений, вы должны быть в состоянии каким-то образом заново включать высокоточное измерение времени (скажем, путем установки `privacy.reduceTimerPrecision` в `false` в Firefox).

11.4.2. Арифметические действия с датами

Объекты `Date` можно сравнивать с помощью стандартных операций сравнения <, <=, > и >= языка JavaScript. Кроме того, один объект `Date` можно вычитать из другого, чтобы определить количество миллисекунд между двумя датами. (Вычитание работает благодаря тому, что в классе `Date` определен метод `valueOf()`, который возвращает отметку времени.)

Если вы хотите добавить или вычесть указанное количество секунд, минут или часов из объекта `Date`, то часто проще модифицировать отметку времени, как демонстрировалось в предыдущем примере, когда мы добавляли к дате 30 секунд. Такой прием становится более громоздким, когда нужно добавлять дни, и вообще не работает для месяцев и годов, поскольку они имеют варьирующееся количество дней. Чтобы выполнять арифметические действия, включающие в себя дни, месяцы и годы, можно применять методы `setDate()`, `setMonth()` и `setYear()`. Скажем, ниже показан код, который добавляет к текущей дате три месяца и две недели:

```
let d = new Date();
d.setMonth(d.getMonth() + 3, d.getDate() + 14);
```

Методы установки дат работают корректно даже в случае переполнения. Когда мы добавляем три месяца к текущему месяцу, то можем получить зна-

чение, превышающее 11 (которое представляет декабрь). Метод `setMonth()` обрабатывает такую ситуацию, инкрементируя год. Аналогично, когда мы устанавливаем день месяца в значение, которое больше количества дней в месяце, то месяц надлежащим образом инкрементируется.

11.4.3. Форматирование и разбор строк с датами

Если вы используете класс `Date` для отслеживания даты и времени (как противоположность измерению временных интервалов), тогда вероятно вам необходимо отображать значения даты и времени пользователям вашего кода. В классе `Date` определено несколько методов для преобразования объектов `Date` в строки. Далее приведены примеры:

```
let d = new Date(2020, 0, 1, 17, 10, 30); // 5:10:30pm, день Нового
                                          // года 2020
d.toString()                // => "Wed Jan 01 2020 17:10:30 GMT-0800
                            // (Pacific Standard Time)"
d.toUTCString()             // => "Thu, 02 Jan 2020 01:10:30 GMT"
d.toLocaleDateString()      // => "1/1/2020": локаль 'en-US'
d.toLocaleTimeString()      // => "5:10:30 PM": локаль 'en-US'
d.toISOString()             // => "2020-01-02T01:10:30.000Z"
```

Ниже представлен полный список методов строкового форматирования в классе `Date`.

- **`toString()`**. Этот метод применяет локальный часовой пояс, но не форматирует дату и время с учетом локали.
- **`toUTCString()`**. Этот метод использует часовой пояс UTC, но не форматирует дату и время с учетом локали.
- **`toISOString()`**. Этот метод выводит дату и время в формате год-месяц-день часы:минуты:секунды.миллисекунды стандарта ISO-8601. Буква "T" в выводе отделяет порцию даты от порции времени. Время выражается в UTC, на что указывает буква "Z" в конце вывода.
- **`toLocaleString()`**. Этот метод применяет локальный часовой пояс и формат, соответствующий локали пользователя.
- **`toDateString()`**. Этот метод форматирует только порцию даты объекта `Date`, опуская время. Он использует локальный часовой пояс и не выполняет форматирование, соответствующее локали.
- **`toLocaleDateString()`**. Этот метод форматирует только дату. Он применяет локальный часовой пояс и формат, соответствующий локали.
- **`toTimeString()`**. Этот метод форматирует только время, опуская дату. Он использует локальный часовой пояс, но не выполняет форматирование времени, соответствующее локали.
- **`toLocaleTimeString()`**. Этот метод форматирует время в соответствии с локалью и применяет локальный часовой пояс.

Ни один из перечисленных выше методов преобразования даты в строку не идеален при форматировании даты и времени для отображения конечным пользователям. Более универсальные и учитывающие локаль приемы форматирования даты и времени описаны в подразделе 11.7.2.

Наконец, в дополнение к методам, преобразующим объект `Date` в строку, имеется также статический метод `Date.parse()`, который принимает в своем аргументе строку, пытается разобрать ее как дату и время и возвращает отметку времени, представляющую эту дату. Метод `Date.parse()` способен разбирать такие же строки, как и конструктор `Date()`, и гарантированно может проводить разбор вывода методов `toISOString()`, `toUTCString()` и `toString()`.

11.5. Классы ошибок

Операторы `throw` и `catch` языка JavaScript могут генерировать и перехватывать любое значение JavaScript, включая элементарные значения. Не существует типа исключения, который должен использоваться для сигнализации об ошибках. Тем не менее, в JavaScript определен класс `Error` и по традиции экземпляры `Error` или его подкласса применяются для сигнализации об ошибке посредством `throw`. Одна из веских причин использовать объект `Error` связана с тем, что при его создании он захватывает состояние стека JavaScript, и если исключение не перехвачено, то стек будет отображаться вместе с сообщением об ошибке, содействуя отладке проблемы. (Следует отметить, что трассировка стека показывает, где был создан объект `Error`, а не где он генерировался оператором `throw`. Если вы всегда создаете объект `Error` прямо перед его генерацией с помощью `throw new Error()`, тогда никакой путаницы не возникнет.)

Объекты `Error` имеют два свойства, `message` и `name`, а также метод `toString()`. Значением свойства `message` будет значение, переданное конструктору `Error()`, которое при необходимости преобразуется в строку. Для объектов ошибок, созданных посредством `Error()`, свойство `name` всегда содержит `"Error"`. Метод `toString()` просто возвращает значение свойства `name`, за которым следует двоеточие, пробел и значение свойства `message`.

Хотя это не входит в стандарт ECMAScript, но среда Node и все современные браузеры определяют в объектах `Error` также свойство `stack`. Значением свойства `stack` является многострочная строка, которая содержит трассировку стека вызовов JavaScript в момент, когда объект `Error` был создан. Такую информацию может быть полезно регистрировать в случае возникновения непредвиденной ошибки.

Кроме класса `Error` в JavaScript определено несколько подклассов, которые применяются для сигнализации об ошибках специфических типов, устанавливаемых стандартом ECMAScript: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` и `URIError`. Вы можете использовать указанные подклассы ошибок в своем коде там, где они выглядят подходящими. Подобно базовому классу `Error` каждый подкласс имеет конструктор, принимающий единственный аргумент сообщения. И экземпляры каждого подкласса имеют свойство `name`, значение которого совпадает с именем конструктора.

Вы можете свободно определять собственные подклассы Error, которые наилучшим образом инкапсулируют условия ошибок в вашей программе. Обратите внимание, что вы не ограничены свойствами name и message. В случае создания подкласса вы можете определять новые свойства для предоставления деталей об ошибке. Скажем, если вы реализуете анализатор, то можете посчитать удобным определить класс ParseError со свойствами line и column, указывающими точное местоположение отказа при разборе. Или при работе с HTTP-запросами вы можете определить класс HTTPError, который имеет свойство status, хранящее код состояния HTTP (такой как 404 или 500) отказавшего запроса. Например:

```
class HTTPError extends Error {
   constructor(status, statusText, url) {
      super(`${status} ${statusText}: ${url}`);
      this.status = status;
      this.statusText = statusText;
      this.url = url;
   }
   get name() { return "HTTPError"; }
}
let error = new HTTPError(404, "Not Found", "http://example.com/");
error.status       // => 404
error.message      // => "404 Not Found: http://example.com/"
error.name         // => "HTTPError"
```

11.6. Сериализация и разбор данных в формате JSON

Когда программе необходимо сохранять данные либо передавать их через сетевое подключение другой программе, она должна преобразовать структуры данных внутри памяти в строку байтов или символов, которые можно сохранить или передать и позже произвести разбор с целью восстановления первоначальных структур данных внутри памяти. Такой процесс преобразования структур данных в потоки байтов или символов называется *сериализацией* (или маршализацией либо даже *складированием*).

Простейший способ сериализации данных в JavaScript предусматривает применение формата JSON. Аббревиатура JSON означает "JavaScript Object Notation" (представление объектов JavaScript) и, как вытекает из названия, формат JSON использует синтаксис литералов типа объектов и массивов JavaScript для преобразования структур данных, состоящих из объектов и массивов, в строки. JSON поддерживает элементарные числа и строки плюс значения true, false и null, а также массивы и объекты, построенные из таких элементарных значений. JSON не поддерживает другие типы JavaScript вроде Map, Set, RegExp, Date или типизированных массивов. Однако он оказался удивительно универсальным форматом данных и широко применяется с программами, не основанными на JavaScript.

В JavaScript поддерживается сериализация и десериализация JSON через две функции, `JSON.stringify()` и `JSON.parse()`, которые кратко раскрывались в разделе 6.8. Имея объект или массив (вложенный произвольно глубоко), который не содержит несериализируемых значений наподобие объектов `RegExp` или типизированных массивов, вы можете сериализировать его, просто передав `JSON.stringify()`. Как следует из имени, эта функция возвращает строку. И располагая строкой, возвращенной `JSON.stringify()`, вы можете воссоздать исходную структуру данных, передав строку `JSON.parse()`:

```
let o = {s: "", n: 0, a: [true, false, null]};
let s = JSON.stringify(o);//s == '{"s":"","n":0,"a":[true,false,null]}'
let copy = JSON.parse(s); //copy == {s:"", n:0, a:[true, false, null]}
```

Если мы пропустим часть, где сериализированные данные сохраняются или посылаются по сети, то можем использовать эту пару функций как несколько неэффективный способ создания глубокой копии объекта:

```
// Создать глубокую копию любого сериализируемого объекта или массива
function deepcopy(o) {
   return JSON.parse(JSON.stringify(o));
}
```

JSON — подмножество JavaScript

Когда данные сериализируются в формат JSON, результат будет допустимым исходным кодом JavaScript для выражений, которые вычисляются в копию первоначальной структуры данных. Если вы снабдите строку JSON префиксом `var data =` и передадите результат функции `eval()`, то получите копию исходной структуры данных, присвоенную переменной `data`. Тем не менее, вы никогда не должны так поступать, потому что возникнет огромная брешь в безопасности — если злоумышленник сможет внедрять произвольный код JavaScript в файл JSON, то сумеет заставить вашу программу выполнить этот код. Быстрее и безопаснее просто применять `JSON.parse()` для декодирования данных в формате JSON.

Временами JSON используется в качестве формата конфигурационных файлов, пригодного для восприятия человеком. Если вы обнаружите, что вручную редактируете файл JSON, то обратите внимание, что формат JSON является очень ограниченным подмножеством JavaScript. Комментарии не разрешены и имена свойств должны заключаться в двойные кавычки, даже когда JavaScript этого не требует.

Обычно вы передаете `JSON.stringify()` и `JSON.parse()` только один аргумент. Обе функции принимают необязательный второй аргумент, который позволяет расширять формат JSON, и он будет описан далее в главе. Функция `JSON.stringify()` также принимает необязательный третий аргумент, который мы обсудим первым. Если вы хотите, чтобы ваша строка в формате JSON была читабельной для человека (скажем, когда она применяется в конфигурационном файле), тогда должны передавать `null` как второй аргумент и число или

строку как третий. Третий аргумент сообщает функции JSON.stringify() о том, что она обязана форматировать данные во множестве строк с отступами. Если в третьем аргументе передано число, то оно будет использоваться в качестве количества пробелов для каждого уровня отступов. Если в третьем аргументе передана строка пробельных символов (таких как '\t'), тогда она будет применяться для каждого уровня отступов.

```
let o = {s: "test", n: 0};
JSON.stringify(o, null, 2)  // => '{\n "s": "test",\n "n": 0\n}'
```

Функция JSON.parse() игнорирует пробельные символы, поэтому передача третьего аргумента функции JSON.stringify() никак не влияет на возможность преобразования строки обратно в структуру данных.

11.6.1. Настройка JSON

Если у функции JSON.stringify() запрашивается сериализация значения, которое естественным образом не поддерживается форматом JSON, то она проверяет, имеется ли у значения метод toJSON(), и при его наличии вызывает данный метод, после чего преобразует в строку возвращаемое значение вместо первоначального значения. Объекты Date реализуют метод toJSON(): он возвращает ту же строку, что и метод toISOString(). Это означает, что если вы сериализируете объект, который включает объект Date, то дата автоматически преобразуется в строку. При разборе сериализированной строки воссозданная структура данных окажется не точно той же самой, с которой вы начинали, поскольку она будет иметь строку там, где исходный объект имел Date.

Если вам нужно воссоздать объекты Date (или по-другому модифицировать разобранные объекты), тогда можете передать во втором аргументе JSON.parse() функцию "восстановления". В случае указания функция восстановления вызывается по одному разу для каждого элементарного значения (но не для объектов или массивов, содержащих такие элементарные значения), разобранного из входной строки. Функция восстановления вызывается с двумя аргументами. Первым аргументом является имя свойства — либо имя свойства объекта, либо индекс массива, преобразованный в строку. Второй аргумент — элементарное значение этого свойства объекта или элемента массива. Кроме того, функция вызывается как метод объекта или массива, который содержит элементарное значение, так что ссылаться на вмещающий объект можно через ключевое слово this.

Возвращаемое значение функции восстановления становится новым значением именованного свойства. Если она возвращает свой второй аргумент, то свойство останется неизмененным. Если она возвращает undefined, тогда именованное свойство будет удалено из объекта или массива до возвращения из JSON.parse().

Ниже показан пример вызова JSON.parse(), в котором используется функция восстановления для фильтрации ряда свойств и воссоздания объектов Date:

```
let data = JSON.parse(text, function(key, value) {
    // Удалить любые значения, чьи имена свойств начинаются
    // с символа подчеркивания
    if (key[0] === "_") return undefined;

    // Если значение - строка в формате даты ISO 8601,
    // тогда преобразовать ее в объект Date
    if (typeof value === "string" &&
        /^\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\dZ$/.test(value)) {
        return new Date(value);
    }

    // В противном случае возвратить значение без изменений
    return value;
});
```

В дополнение к описанному ранее применению в `toJSON()` функция `JSON.stringify()` также позволяет настраивать свой выход за счет передачи в необязательном втором параметре массива или функции.

Если во втором аргументе передается массив строк (или чисел, которые преобразуются в строки), тогда они используются в качестве имен свойств объекта (или элементов массива). Любое свойство, имя которого отсутствует в массиве, будет исключено из строкового преобразования. Вдобавок возвращаемая строка будет содержать свойства в том же порядке, в каком они указаны в массиве (что может быть очень удобным при написании тестов).

Если вы передаете функцию, то она будет функцией замещения — фактически инверсией необязательной функции восстановления, которую можно передавать `JSON.parse()`. В случае указания функция замещения вызывается для каждого значения, подлежащего строковому преобразованию. В первом аргументе функции замещения передается имя свойства объекта либо индекс массива значения внутри этого объекта, а во втором аргументе — само значение. Функция замещения вызывается как метод объекта или массива, который содержит значение, преобразуемое в строку. Возвращаемое значение функции замещения представляет собой преобразованное в строку первоначальное значение. Если функция замещения возвращает `undefined` или вообще ничего не возвращает, тогда это значение (и его элемент массива или свойство объекта) исключается из строкового преобразования.

```
// Указать, какие поля должны сериализироваться и в каком порядке
let text = JSON.stringify(address, ["city","state","country"]);

// Указать функцию замещения, которая опускается свойства
// со значениями RegExp
let json=JSON.stringify(o, (k,v)=> v instanceof RegExp?undefined:v);
```

Два вызова `JSON.stringify()` здесь используют второй аргумент щадящим образом, производя сериализированный вывод, который может быть десериализирован без специальной функции восстановления. Однако в целом, если вы определяете метод `toJSON()` для типа или применяете функцию замещения, которая на самом деле заменяет несериализируемые значения сериализируемыми, тогда обычно для получения первоначальной структуры данных в вызове

JSON.parse() придется использовать специальную функцию восстановления. Поступая так, вы должны осознавать, что определяете специальный формат данных, принося в жертву переносимость и совместимость с огромной экосистемой совместимых с JSON инструментов и языков.

11.7. API-интерфейс интернационализации

API-интерфейс интернационализации JavaScript состоит из трех классов Intl.NumberFormat, Intl.DateTimeFormat и Intl.Collator, которые позволяют форматировать числа (включая денежные суммы и процентные отношения), дату и время, а также сравнивать сроки способами, соответствующими локали. Указанные классы не являются частью стандарта ECMAScript, но определены в рамках стандарта ECMA402 (https://tc39.es/ecma402/) и эффективно поддерживаются веб-браузерами. API-интерфейс Intl также поддерживается в Node, но на момент написания главы двоичные сборки Node не поставлялись с данными локализации, которые требуются для работы с локалями, отличающимися от US English. Таким образом, для применения классов интернационализации вместе с Node может понадобится загрузить отдельный пакет данных либо использовать специальную сборку Node.

Одной из наиболее важных частей интернационализации является отображение текста, который был переведен на язык пользователя. Достичь цели можно различными путями, но они выходят за рамки рассматриваемого здесь API-интерфейса Intl.

11.7.1. Форматирование чисел

Пользователи во всем мире ожидают, что числа будут форматироваться разнообразными способами. Десятичные точки могут быть точками или запятыми. Разделители тысяч могут быть запятыми или точками и не применяться через каждые три цифры во всех местах. Одни валюты поддерживают сотые части, другие — тысячные части, а третьи не имеют дроблений. Наконец, хотя так называемые "арабские цифры" от 0 до 9 используются во многих языках, они не универсальны, и пользователи в некоторых странах ожидают видеть числа представленными в принятых у них системах письма.

В классе Intl.NumberFormat определен метод format(), который учитывает все эти возможности форматирования. Его конструктор принимает два аргумента. Первый аргумент указывает локаль, для которой должно быть сформатировано число, а второй — объект, предоставляющий дополнительные детали о том, как форматировать число. Если первый аргумент опущен или равен undefined, тогда будет применяться системная локаль (исходя из предположения, что она предпочтительна для пользователя). Если в первом аргументе передается строка, то она указывает желательную локаль, такую как "en-US" (английский в США), "fr" (французский) или "zh-Hans-CN" (китайский в Китае, использующий упрощенную систему письма). Первым аргументом также может быть массив строк локалей, и в таком случае класс Intl.NumberFormat будет выбирать наиболее специфическую из них, которая хорошо поддерживается.

Вторым аргументом конструктора `Intl.NumberFormat()`, если он передан, должен быть объект, который определяет одно или большее количество перечисленных далее свойств.

- **`style`**. Это свойство задает требуемый стиль форматирования чисел. По умолчанию принимается `"decimal"`. Укажите `"percent"` для форматирования числа как процентного отношения или `"currency"`, чтобы представить число как денежную сумму.
- **`currency`**. В случае стиля `"currency"` это свойство необходимо для указания трехбуквенного кода валюты ISO (такого как `"USD"` для долларов США или `"GBP"` для английских фунтов).
- **`currencyDisplay`**. В случае стиля `"currency"` это свойство указывает, как отображается валюта. Принятое по умолчанию значение `"symbol"` приводит к применению символа валюты, если валюта им располагает. Значение `"code"` вызывает использование трехбуквенного кода ISO, а значение `"name"` — названия валюты в длинной форме.
- **`useGrouping`**. Установите это свойство в `false`, если не хотите, чтобы числа имели разделители тысяч (либо их эквиваленты в локали).
- **`minimumIntegerDigits`**. Минимальное количество цифр для отображения в целой части числа. Если число имеет меньше цифр, чем минимальное количество, тогда оно дополняется слева нулями. По умолчанию принимается 1, но можно применять значения вплоть до 21.
- **`minimumFractionDigits`**, **`maximumFractionDigits`**. Эти два свойства управляют форматирование дробной части числа. Если число имеет меньше цифр в дробной части, чем минимум, тогда оно будет дополнено справа нулями. Если цифр в дробной части больше, чем максимум, то дробная часть округляется. Допустимые значения обоих свойств находятся между 0 и 20. По умолчанию для минимума принимается 0, а для максимума — 3, исключая форматирование денежных сумм, когда длина дробной части варьируется в зависимости от указанной валюты.
- **`minimumSignificantDigits`**, **`maximumSignificantDigits`**. Эти два свойства управляют количеством значащих цифр, используемых при форматировании числа, что делает их подходящими в случае форматирования научных данных, например. Когда указаны, свойства `minimumSignificantDigits` и `maximumSignificantDigits` переопределяют ранее описанные свойства `minimumFractionDigits` и `maximumFractionDigits`. Допустимые значения обоих свойств находятся между 1 и 21.

После создания объекта `Intl.NumberFormat` с желательной локалью и параметрами вы можете задействовать этот объект, передавая число его методу `format()`, который возвращает надлежащим образом сформатированную строку.

Вот пример:

```
let euros = Intl.NumberFormat("es", {style: "currency", currency: "EUR"});
euros.format(10)    // => "10,00 €": десять евро, испанское форматирование

let pounds = Intl.NumberFormat("en", {style: "currency", currency: "GBP"});
pounds.format(1000) // => "£1,000.00": одна тысяча фунтов,
                    //                 английское форматирование
```

Полезной особенностью класса `Intl.NumberFormat` (и других классов `Intl`) является то, что его метод `format()` привязан к объекту `NumberFormat`, которому он принадлежит. Таким образом, вместо определения переменной, ссылающейся на объект форматирования, и затем вызова на ней метода `format()` вы можете просто присвоить метод `format()` переменной и применять ее, как если бы она была автономной функцией:

```
let data = [0.05, .75, 1];
let formatData = Intl.NumberFormat(undefined, {
   style: "percent",
   minimumFractionDigits: 1,
   maximumFractionDigits: 1
}).format;

data.map(formatData) // => ["5.0%", "75.0%", "100.0%"]: в локали en-US
```

Некоторые языки наподобие арабского используют собственную систему письма для десятичных цифр:

```
let arabic = Intl.NumberFormat("ar", {useGrouping: false}).format;
arabic(1234567890)    // => "١٢٣٤٥٦٧٨٩٠"
```

Другие языки, такие как хинди, применяют систему письма, которая имеет собственный набор цифр, но по умолчанию обычно используют цифры ASCII от 0 до 9. Если вы хотите переопределить стандартную систему письма, применяемую для цифр, тогда добавьте `-u-nu-` к локали и затем укажите сокращенное название системы письма. Скажем, вы можете форматировать числа с группированием в индийском стиле и цифрами деванагари:

```
let hindi = Intl.NumberFormat("hi-IN-u-nu-deva").format;
hindi(1234567890)     // => "१,२३,४५,६७,८९०"
```

`-u-` в локали указывает, что далее следует расширение Unicode. `nu` — название расширения для системы счисления, а `deva` — сокращение для деванагари (Devanagari). В стандарте API-интерфейса `Intl` определены имена для ряда других систем счисления, главным образом для индоарийских языков Южной и Юго-Восточной Азии.

11.7.2. Форматирование даты и времени

Класс `Intl.DateTimeFormat` во многом похож на класс `Intl.NumberFormat`. Конструктор `Intl.DateTimeFormat()` принимает такие же два аргумента, как `Intl.NumberFormat()`: локаль или массив локалей и объект параметров форматирования. И способ использования экземпляра `Intl.DateTimeFormat` аналогичен: вызов его метода `format()` для преобразования объекта `Date` в строку.

Как упоминалось в разделе 11.4, в классе Date определены простые методы toLocaleDateString() и toLocaleTimeString(), которые производят вывод, соответствующий локали пользователя. Но эти методы не дают вам никакого контроля над тем, какие поля даты и времени отображаются. Может быть, вы хотите опустить год, но добавить день недели к формату даты. Как нужно представить месяц — в виде его номера или названия? Класс Intl.DateTimeFormat обеспечивает детальный контроль над тем, что выводится, на основе свойств объекта параметров, который передается конструктору во втором аргументе. Тем не менее, обратите внимание, что Intl.DateTimeFormat не всегда способен отображать в точности то, что вы запросили. Если вы укажете параметры для форматирования часов и секунд, но опустите минуты, то обнаружите, что объект Intl.DateTimeFormat все равно отображает минуты. Идея заключается в том, что вы применяете объект параметров для указания, какие поля даты и времени желательно представлять пользователю, и как их желательно форматировать (скажем, по названию или по числу), после чего объект Intl.DateTimeFormat будет искать формат, согласованный с локалью, который наиболее близко соответствует вашим запросам.

Ниже перечислены доступные параметры. Указывайте только те свойства для даты и времени, которые вы хотели бы отображать в сформатированном выводе.

- **year**. Используйте "numeric" для полного представления года из четырех цифр или "2-digit" для сокращения из двух цифр.
- **month**. Применяйте "numeric" для возможно коротких номеров вроде "1" или "2-digit" для числового представления, которое всегда содержит два цифры, как "01". Используйте "long" для полного названия наподобие "January", "short" для сокращения вроде "Jan" и "narrow" для сильно сокращенного названия, подобного "J", которое не будет гарантированно уникальным.
- **day**. Применяйте "numeric" для номера из одной или двух цифр или "2-digit" для номера дня месяца из двух цифр.
- **weekday**. Используйте "long" для полного названия, такого как "Monday", "short" для сокращенного названия вроде "Mon" и "narrow" для сильно сокращенного названия наподобие "M", которое не будет гарантированно уникальным.
- **era**. Это свойство указывает, должна ли дата форматироваться с эрой, такой как CE (н.э.) или BCE (до н.э.). Оно может быть полезно при форматировании дат из очень давних времен или в случае применения японского календаря. Допустимые значения — "long", "short" и "narrow".
- **hour**, ***minute***, ***second***. Эти свойства указывают, как желательно отображать время. Используйте "numeric" для поля из одной или двух цифр или "2-digit", чтобы принудительно дополнять слева нулем числа с одной цифрой.

- **timeZone**. Это свойство указывает желательный часовой пояс, для которого должна форматироваться дата. Если опущено, тогда применяется локальный часовой пояс. Реализации всегда распознают "UTC" и могут также распознавать названия часовых поясов IANA (Internet Assigned Numbers Authority — Комитет по цифровым адресам в Интернете), такие как "America/Los_Angeles".
- **timeZoneName**. Это свойство указывает, каким образом часовой пояс должен отображаться в сформатированной дате или времени. Используйте `"long"` для полного названия часового пояса и `"short"` для сокращенного или числового часового пояса.
- **hour12**. Это булевское свойство указывает, применяется ли 12-часовое время. По умолчанию принимается параметр из локали, но вы можете переопределить его с помощью данного свойства.
- **hourCycle**. Это свойство позволяет указать, каким образом записывается полночь — 0 часов, 12 часов или 24 часа. По умолчанию принимается параметр из локали, но вы можете переопределить его с помощью данного свойства. Обратите внимание, что `hour12` имеет преимущество перед свойством `hourCycle`. Используйте `"h11"` для указания на то, что полночь — 0, а час перед полночью — 11pm. Применяйте `"h12"` для указания на то, что полночь — 12. Используйте `"h23"` для указания на то, что полночь — 0, а час перед полночью — 23. И применяйте `"h24"` для указания на то, что полночь — 24.

Вот несколько примеров:

```
let d = new Date("2020-01-02T13:14:15Z");          //2 января 2020 года,
                                                   // 13:14:15 UTC

// Без параметров мы получаем базовый числовой формат даты
Intl.DateTimeFormat("en-US").format(d)             // => "1/2/2020"
Intl.DateTimeFormat("fr-FR").format(d)             // => "02/01/2020"

// Выводить день недели и месяц
let opts={weekday:"long",month:"long",year:"numeric",day: "numeric"};
Intl.DateTimeFormat("en-US",opts).format(d) //=>"Thursday, January 2, 2020"
Intl.DateTimeFormat("es-ES",opts).format(d) //=>"jueves, 2 de enero de 2020"

// Время в Нью-Йорке для франкоговорящих канадцев
opts={hour: "numeric",minute:"2-digit",timeZone:"America/New_York"};
Intl.DateTimeFormat("fr-CA", opts).format(d)       // => "8 h 14"
```

Класс `Intl.DateTimeFormat` способен отображать даты с использованием календарей, которые отличаются от стандартного юлианского календаря, основанного на нашей эре. Хотя некоторые локали по умолчанию могут применять не юлианский календарь, у вас всегда есть возможность явно указать используемый календарь, добавив к локали `-u-ca-` и название календаря. Допустимые названия календарей включают "buddhist" (буддистский), "chinese" (китайский), "coptic" (коптский), "ethiopic" (эфиопский), "gregory" (григорианский), "hebrew" (еврейский), "indian" (индийский), "islamic" (исламский), "iso8601" (ISO 8601),

"japanese" (японский) и "persian" (персидский). Продолжая предыдущий пример, мы можем определить год в различных не юлианских календарях:

```
let opts = { year: "numeric", era: "short" };
Intl.DateTimeFormat("en", opts).format(d)              // => "2020 AD"
Intl.DateTimeFormat("en-u-ca-iso8601", opts).format(d) // => "2020 AD"
Intl.DateTimeFormat("en-u-ca-hebrew", opts).format(d)  // => "5780 AM"
Intl.DateTimeFormat("en-u-ca-buddhist", opts).format(d) // => "2563 BE"
Intl.DateTimeFormat("en-u-ca-islamic", opts).format(d) // => "1441 AH"
Intl.DateTimeFormat("en-u-ca-persian", opts).format(d) // => "1398 AP"
Intl.DateTimeFormat("en-u-ca-indian", opts).format(d)  // => "1941 Saka"
Intl.DateTimeFormat("en-u-ca-chinese", opts).format(d) // => "36 78"
Intl.DateTimeFormat("en-u-ca-japanese", opts).format(d) // => "2 Reiwa"
```

11.7.3. Сравнение строк

Задача сортировки строк в алфавитном порядке (или в каком-то более общем "порядке сопоставления" для неалфавитных систем письма) сложнее, чем часто представляют себе англоязычные. В английском языке применяется относительно небольшой алфавит без букв с диакритическими знаками, и мы имеем преимущество кодировки символов (ASCII с тех пор, как она включена в Unicode), чьи числовые значения полностью соответствуют стандартному порядку сортировки строк. В других языках все не так просто. Скажем, в испанском ñ трактуется как отдельная буква, находящаяся после n и перед o. Литовский язык располагает в алфавитном порядке Y перед J, а валлийский язык трактует пары букв вроде CH и DD как одиночные буквы, причем CH идет после C и DD сортируется после D.

Если вы хотите отображать строки пользователю в порядке, который он сочтет естественным, тогда использования метода `sort()` на массиве строк будет недостаточно. Но если вы создадите объект `Intl.Collator`, то сможете передать методу `sort()` метод `compare()` этого объекта, чтобы выполнять сортировку строк в соответствии с локалью. Объекты `Intl.Collator` можно конфигурировать так, что метод `compare()` будет выполнять сравнения, нечувствительные к регистру, или даже сравнения, которые учитывают только базовую букву и игнорируют ударения и другие диакритические знаки.

Подобно `Intl.NumberFormat()` и `Intl.DateTimeFormat()` конструктор `Intl.Collator()` принимает два аргумента. В первом аргументе передается локаль или массив локалей, а во втором — необязательный объект, свойства которого указывают, какой в точности вид сравнения строк должен делаться. Ниже описаны поддерживаемые свойства.

- **usage**. Это свойство указывает, как необходимо применять объект сопоставления. По умолчанию принимается значение `"sort"`, но вы можете также указать `"search"`. Идея в том, что при сортировке строк вам обычно нужен объект сопоставления, который различает как можно больше строк для обеспечения надежного упорядочения. Но при сравнении двух строк некоторые локали могут требовать менее строгого сравнения, которое игнорирует знаки ударения, например.

- **sensitivity**. Это свойство указывает, чувствителен ли объект сопоставления к регистру букв и знакам ударения при сравнении строк. Значение "base" вызывает сравнения, которые игнорируют регистр и знаки ударения, учитывая для каждого символа только базовую букву. (Однако обратите внимание, что некоторые языки считают определенные символы с диакритическими знаками отдельными базовыми буквами.) Значение "accent" учитывает знаки ударения, но игнорирует регистр. Значение "case" учитывает регистр и игнорирует знаки ударения. Значение "variant" выполняет строгие сравнения, которые учитывают регистр и знаки ударения. Стандартным значением для этого свойства является "variant", когда usage равно "sort". Если usage равно "search", тогда чувствительность по умолчанию зависит от локали.
- **ignorePunctuation**. Установите это свойство в true, чтобы игнорировать пробелы и знаки пунктуации при сравнении строк. Когда данное свойство установлено в true, то строки наподобие "any one" и "anyone" будут считаться равными.
- **numeric**. Установите это свойство в true, если сравниваемые строки представляют собой или содержат целые числа, и вы хотите сортировать их в числовом, а не алфавитном порядке. Когда данное свойство установлено в true, то в результате сортировки строка "Version 9" окажется перед строкой "Version 10".
- **caseFirst**. Это свойство указывает, какой регистр букв должен быть первым. Если вы установите его в "upper", тогда "A" при сортировке окажется перед "a". Если же вы установите его в "lower", то "a" при сортировке будет перед "A". В любом случае имейте в виду, что варианты буквы в верхнем и нижнем регистре будут располагаться рядом друг с другом в порядке сортировки, который отличается от лексикографического порядка Unicode (стандартное поведение метода sort() класса Array), где все буквы ASCII верхнего регистра находятся перед всеми буквами ASCII нижнего регистра. Принимаемое по умолчанию значение для данного свойства зависит от локали, а реализации могут его игнорировать и не разрешать переопределение порядка сортировки регистра.

После создания объекта Intl.Collator с желаемой локалью и параметрами вы можете использовать его метод compare() для сравнения двух строк. Данный метод возвращает число. Если возвращаемое значение меньше нуля, тогда первая строка идет перед второй строкой. Если возвращаемое значение больше нуля, тогда первая строка идет после второй строки. Если метод compare() возвращает ноль, тогда с точки зрения этого объекта сопоставления две строки равны.

Такой метод compare(), принимающий две строки и возвращающий число, которое меньше, равно или больше нуля, является в точности тем, что ожидает метод sort() класса Array для своего необязательного аргумента. Кроме того, Intl.Collator автоматически привязывает метод compare() к его экземпля-

ру, так что вы можете передавать его напрямую методу sort() без необходимости в написании функции оболочки и ее вызове через объект сопоставления. Ниже приведены примеры:

```
// Базовый компаратор для сортировки согласно локали пользователя.
// Никогда не сортируйте строки, читабельные человеком, без передачи
чего-то вроде такого:
const collator = new Intl.Collator().compare;
["a", "z", "A", "Z"].sort(collator)       // => ["a", "A", "z", "Z"]

// Имена файлов часто включают числа, поэтому мы должны их
// сортировать особым образом:
const filenameOrder = new Intl.Collator(undefined,
                                        { numeric: true }).compare;
["page10", "page9"].sort(filenameOrder)  // => ["page9", "page10"]
// Найти все строки, которые примерно соответствуют целевой строке:
const fuzzyMatcher = new Intl.Collator(undefined, {
   sensitivity: "base",
   ignorePunctuation: true
}).compare;
let strings = ["food", "fool", "Fuш Bar"];
strings.findIndex(s => fuzzyMatcher(s, "foobar") === 0)  // => 2
```

Некоторые локали имеют более одного возможного порядка сопоставления. Скажем, в телефонных справочниках в Германии применяется чуть более фонетический порядок сортировки, чем в словарях. В Испании до 1994 года "ch" и "ll" считались отдельными буквами, а теперь в этой стране существует современный и традиционный порядки сортировки. В Китае порядок сопоставления может быть основан на кодировках символов, базовом корне и штрихах каждого символа или романизации пиньинь символов. Такие варианты сопоставления нельзя выбирать посредством аргумента параметров Intl.Collator, но можно выбрать путем добавления к строке локали -u-co- и названия желаемого варианта. Например, используйте "de-DE-u-co-phonebk" для порядка, принятого для телефонных справочников в Германии, и "zh-TW-u-co-pinyin" для упорядочения пиньинь на острове Тайвань.

```
// До 1994 года в Испании CH и LL считались отдельными буквами
const modernSpanish = Intl.Collator("es-ES").compare;
const traditionalSpanish = Intl.Collator("es-ES-u-co-trad").compare;
let palabras = ["luz", "llama", "como", "chico"];
palabras.sort(modernSpanish)  // => ["chico", "como", "llama", "luz"]
palabras.sort(traditionalSpanish) // => ["como", "chico", "luz", "llama"]
```

11.8. API-интерфейс Console

Вы уже видели, что функция console.log() применяется во многих местах книги: в веб-браузерах она выводит строку на вкладке Console (Консоль) панели инструментов разработчика, что может оказаться очень полезным при отладке. В Node функция console.log() является универсальной функцией вывода и выводит свои аргументы в поток stdout процесса, где они обычно отображаются пользователю в окне терминала как вывод программы.

Помимо `console.log()` в API-интерфейсе Console определено еще несколько полезных функций. Данный API-интерфейс не входит в состав стандарта ECMAScript, но поддерживается браузерами и Node и формально был написан и стандартизирован на веб-сайте `https://console.spec.whatwg.org/`.

В API-интерфейсе Console определены следующие функции.

- **`console.log()`**. Это самая известная консольная функция, которая преобразует свои аргументы в строки и выводит их на консоль. Она включает пробелы между аргументами и переходит на новую строку после вывода всех аргументов.
- **`console.debug()`, `console.info()`, `console.warn()`, `console.error()`**. Эти функции практически идентичны `console.log()`. В Node функция `console.error()` посылает свой вывод в поток `stderr`, а не `stdout`, но остальные функции являются псевдонимами `console.log()`. В браузерах выходные сообщения, генерируемые каждой функцией, могут предваряться значком, который указывает уровень их серьезности, и консоль разработчика может также позволять разработчикам фильтровать консольные сообщения по уровню серьезности.
- **`console.assert()`**. Если первый аргумент истинный (т.е. утверждение проходит), тогда эта функция ничего не делает. Но когда в первом аргументе передается `false` или другое ложное значение, то оставшиеся аргументы выводятся, как если бы они были переданы функции `console.error()` с префиксом "Assertion failed" (Отказ утверждения). Обратите внимание, что в отличие от типичных функций `assert()` в случае отказа утверждения функция `console.assert()` не генерирует исключение.
- **`console.clear()`**. Данная функция очищает консоль, когда это возможно. Она работает в браузерах и Node при отображении средой Node своего вывода в окне терминала. Тем не менее, если вывод Node перенаправляется в файл или канал, то вызов `console.clear()` ничего не делает.
- **`console.table()`**. Эта функция является необыкновенно мощным, но малоизвестным инструментом для выработки табличного вывода, который особенно удобен в программах Node, нуждающихся в выдаче вывода с подытоженными данными. Функция `console.table()` пытается отобразить свои аргументы в табличной форме (и если ей это не удается, то она отображает данные с использованием обычного форматирования `console.log()`). Функция `console.table()` работает лучше всего, когда аргумент представляет собой относительно короткий массив объектов, и все объекты в массиве имеют один и тот же (относительно небольшой) набор свойств. В таком случае каждый объект в массиве форматируется в виде строки таблицы, причем каждое свойство становится столбцом таблицы. Можно также передать в необязательном втором аргументе массив имен свойств, чтобы указать желаемый набор столбцов. Если вместо массива объектов передан единственный объект, тогда на выходе будет таблица со столбцами для имен свойств и одной строкой для значений свойств. Если сами значения свойств являются объектами, то имена их свойств станут столбцами в таблице.

- **`console.trace()`**. Эта функция выводит свои аргументы подобно `console.log()` и вдобавок сопровождает вывод информацией трассировки стека. В Node вывод направляется в `stderr`, а не в `stdout`.
- **`console.count()`**. Функция `console.count()` принимает строковый аргумент и выводит эту строку вместе с количеством ее вызовов с данной строкой. Она может быть полезна, скажем, при отладке обработчика событий, если необходимо отслеживать, сколько раз обработчик событий запускался.
- **`console.countReset()`**. Эта функция принимает строковый аргумент и сбрасывает счетчик для данной строки.
- **`console.group()`**. Эта функция выводит свои аргументы на консоль, как если бы они были переданы функции `console.log()`, и затем устанавливает внутреннее состояние консоли, так что все последующие консольные сообщения (вплоть до вызова `console.groupEnd()`) будут иметь отступы относительно текущего выведенного сообщения. Такой прием позволяет визуально выделять группу связанных сообщений с помощью отступов. Консоль разработчика в веб-браузерах обычно дает возможность сворачивать и разворачивать сгруппированные сообщения. Аргументы функции `console.group()`, как правило, применяются для снабжения группы поясняющим именем.
- **`console.groupCollapsed()`**. Эта функция работает подобно `console.group()`, но в веб-браузерах группа по умолчанию будет "свернутой", а содержащиеся в ней сообщения окажутся скрытыми до тех пор, пока пользователь не выполнит щелчок, чтобы развернуть группу. В Node данная функция является синонимом для `console.group()`.
- **`console.groupEnd()`**. Эта функция не принимает аргументов. Она не производит собственного вывода, но заканчивает отступы и группирование, обусловленные самым последним вызовом `console.group()` или `console.groupCollapsed()`.
- **`console.time()`**. Эта функция принимает единственный строковый аргумент, записывает время своего вызова с данной строкой и ничего не выводит.
- **`console.timeLog()`**. Эта функция принимает строку в качестве своего первого аргумента. Если такая строка ранее передавалась методу `console.time()`, тогда функция выводит ее вместе со временем, которое прошло с момента вызова `console.time()`. Любые дополнительные аргументы, указанные при вызове `console.timeLog()`, выводятся, как если бы они передавались функции `console.log()`.
- **`console.timeEnd()`**. Эта функция принимает единственный строковый аргумент. Если такая строка ранее передавалась методу `console.time()`, тогда она выводится вместе с истекшим временем. После вызова `console.timeEnd()` больше не разрешено вызывать `console.timeLog()` без предварительного вызова `console.time()`.

11.8.1. Форматирование вывода с помощью API-интерфейса Console

Консольные функции, которые выводят свои аргументы, вроде `console.log()`, обладают одной малоизвестной особенностью: если первым аргументом является строка, содержащая `%s`, `%i`, `%d`, `%f`, `%o`, `%O` или `%c`, тогда она трактуется как форматная строка[6], а значения последующих аргументов подставляются вместо двухсимвольных последовательностей `%`.

Ниже описано предназначение упомянутых последовательностей.

- **`%s`.** Аргумент преобразуется в строку.
- **`%i`** и **`%d`.** Аргумент преобразуется в число и затем усекается до целого.
- **`%f`.** Аргумент преобразуется в число.
- **`%o`** и **`%O`.** Аргумент трактуется как объект, имена и значения свойств которого выводятся. (В веб-браузерах такое отображение обычно является интерактивным, т.е. пользователи могут развертывать и свертывать свойства с целью исследования вложенной структуры данных.) Последовательности `%o` и `%O` обе отображают детали объекта. Вариант `%O` применяет формат вывода, зависящий от реализации, который считается наиболее полезным для разработчиков программного обеспечения.
- **`%c`.** В веб-браузерах аргумент интерпретируется как строка стилей CSS и используется для стилизации любого идущего после него текста (вплоть до следующей последовательности `%c` или до конца строки). В Node последовательность `%c` и соответствующий ей аргумент попросту игнорируются.

Имейте в виду, что необходимость в применении форматной строки с консольными функциями возникает нечасто: обычно подходящий вывод легче получить, просто передавая одно или большее количество значений (включая объекты) функции и позволяя реализации отобразить их удобным способом. В качестве примера следует отметить, что если вы передадите объект `Error` функции `console.log()`, то она автоматически выведет его вместе с трассировкой стека.

11.9. API-интерфейсы URL

Поскольку JavaScript настолько часто используется в веб-браузерах, в коде JavaScript нужно манипулировать указателями URL. Класс `URL` разбирает URL и также делает возможной модификацию (скажем, добавление параметров поиска или вариантов путей) существующих URL. Кроме того, он надлежащим образом обрабатывает сложные аспекты кодирования и декодирования различных компонентов указателя URL.

[6] Программисты на языке C знакомы со многими из этих символьных последовательностей, т.к. они используются в функции `printf()`.

Класс URL не является частью какого-либо стандарта ECMAScript, но он работает в Node и во всех веб-браузерах кроме Internet Explorer. Класс URL стандартизирован согласно https://url.spec.whatwg.org/.

Объект URL создается с помощью конструктора URL(), которому в качестве аргумента передается строка абсолютного URL. Или же можно передать в первом аргументе относительный URL, во втором — абсолютный URL, относительно которого указан URL в первом аргументе. Когда объект URL создан, его разнообразные свойства позволяют запрашивать включенные версии разных частей URL:

```
let url = new URL("https://example.com:8000/path/name?q=term#fragment");
url.href   // => "https://example.com:8000/path/name?q=term#fragment"
url.origin         // => "https://example.com:8000"
url.protocol       // => "https:"
url.host           // => "example.com:8000"
url.hostname       // => "example.com"
url.port           // => "8000"
url.pathname       // => "/path/name"
url.search         // => "?q=term"
url.hash           // => "#fragment"
```

Хотя так поступают нечасто, указатели URL могут содержать имя пользователя или имя пользователя и пароль, и класс URL тоже способен разбирать эти компоненты:

```
let url = new URL("ftp://admin:1337!@ftp.example.com/");
url.href           // => "ftp://admin:1337!@ftp.example.com/"
url.origin         // => "ftp://ftp.example.com"
url.username       // => "admin"
url.password       // => "1337!"
```

Свойство origin здесь представляет собой простое сочетание протокола и хоста URL (а также порта, если он указан). Как таковое, оно предназначено только для чтения. Но все остальные свойства, продемонстрированные в предыдущем примере, допускают чтение и запись: вы можете устанавливать любое свойство, чтобы устанавливать соответствующую часть URL:

```
let url = new URL("https://example.com"); // Начать с нашего сервера
url.pathname = "api/search";    // Добавить путь к конечной точке API
url.search = "q=test";          // Добавить параметр запроса
url.toString()     // => "https://example.com/api/search?q=test"
```

Одна из важных характеристик класса URL заключается в том, что он корректно добавляет знаки пунктуации и при необходимости отменяет специальные символы в указателях URL:

```
let url = new URL("https://example.com");
url.pathname = "path with spaces";
url.search = "q=foo#bar";
url.pathname       // => "/path%20with%20spaces"
url.search         // => "?q=foo%23bar"
url.href   // => "https://example.com/path%20with%20spaces?q=foo%23bar"
```

Свойство href в приведенных примерах является особенным: чтение href эквивалентно вызову toString() — оно заново собирает все части URL в каноническую строковую форму URL. Установка href в новую строку запускает на новой строке средство разбора URL, как если бы вы снова вызвали конструктор URL().

В предшествующих примерах мы применяли свойство search для ссылки на полную порцию запроса в URL, в состав которой входят символы от знака вопроса и до конца URL или первого символа косой черты. Временами достаточно обращаться с ним как с одиночным свойством URL. Однако часто HTTP-запросы кодируют значения множества полей форм или множества параметров API-интерфейса внутри порции запроса указателя URL, используя формат application/x-www-form-urlencoded. В таком формате порция запроса URL состоит из знака вопроса, за которым следует одна или большее количество пар имя/значение, отделенных друг от друга амперсандами. Одно и то же имя может встречаться несколько раз, давая в результате именованный параметр поиска, который имеет более одного значения.

Если вы хотите закодировать пары имя/значение подобного рода в виде порции запроса URL, тогда свойство searchParams будет удобнее, чем свойство search. Свойство search представляет собой строку для чтения/записи, которая позволяет получать и устанавливать целую порцию запроса URL. Свойство searchParams — это ссылка, допускающая только чтение, на объект URLSearchParams, который имеет API-интерфейс для получения, установки, добавления, удаления и сортировки параметров, закодированных в порции запроса URL:

```
let url = new URL("https://example.com/search");
url.search                          // => "": пока запроса нет
url.searchParams.append("q", "term");  // Добавить параметр поиска
url.search                          // => "?q=term"
url.searchParams.set("q", "x");// Изменить значение этого параметра
url.search                          // => "?q=x"
url.searchParams.get("q")           // => "x": запросить значение параметра
url.searchParams.has("q")           // => true: имеется параметр q
url.searchParams.has("p")           // => false: параметр p отсутствует
url.searchParams.append("opts","1");// Добавить еще один параметр поиска
url.search                          // => "?q=x&opts=1"
url.searchParams.append("opts","&");// Добавить еще одно значение
                                    // для того же самого имени
url.search                          // => "?q=x&opts=1&opts=%26":
                                    //    обратите внимание на отмену
url.searchParams.get("opts")        // => "1": первое значение
url.searchParams.getAll("opts") // => ["1", "&"]: все значения
url.searchParams.sort(); // Разместить параметры в алфавитном порядке
url.search                          // => "?opts=1&opts=%26&q=x"
url.searchParams.set("opts", "y"); // Изменить параметр opts
url.search                          // => "?opts=y&q=x"
// Объект searchParams итерируемый
[...url.searchParams]               // => [["opts", "y"], ["q", "x"]]
url.searchParams.delete("opts"); // Удалить параметр opts
url.search                          // => "?q=x"
url.href                            // => "https://example.com/search?q=x"
```

Значением свойства searchParams является объект URLSearchParams. При желании закодировать параметры URL в строку запроса вы можете создать объект URLSearchParams, добавить параметры, затем преобразовать его в строку и установить ее для свойства search экземпляра URL:

```
let url = new URL("http://example.com");
let params = new URLSearchParams();
params.append("q", "term");
params.append("opts", "exact");
params.toString()      // => "q=term&opts=exact"
url.search = params;
url.href               // => "http://example.com/?q=term&opts=exact"
```

11.9.1. Унаследованные функции для работы с URL

До определения ранее описанного API-интерфейса URL предпринималось много попыток поддержки кодирования и декодирования URL в базовом языке JavaScript. Первой попыткой были глобально определенные функции escape() и unescape(), которые теперь считаются устаревшими, но по-прежнему широко реализованы. Они не должны применяться.

Когда escape() и unescape() стали устаревшими, в стандарте ECMAScript появились две пары альтернативных глобальных функций.

- **encodeURI() и decodeURI()**. Функция encodeURI() принимает в своем аргументе строку и возвращает новую строку, в которой закодированы символы, отличающиеся от ASCII, плюс определенные символы ASCII (такие как пробел). Функция decodeURI() выполняет обратный процесс. Символы, нуждающиеся в кодировании, сначала преобразуются в код UTF-8, затем каждый байт кода заменяется управляющей последовательностью %*XX*, где *XX* — две шестнадцатеричные цифры. Из-за того, что функция encodeURI() предназначена для кодирования полных URL, она не кодирует символы разделителей URL, такие как /, ? и #. Но это означает, что encodeURI() не может работать корректно для URL, которые имеют упомянутые символы внутри своих компонентов.
- **encodeURIComponent() и decodeURIComponent()**. Данные две функции работают подобно encodeURI() и decodeURI(), но они предназначены для кодирования индивидуальных компонентов URI, а потому также кодируют управляющие символы вроде /, ? и #, которые используются для отделения этих компонентов. Из унаследованных функций для работы с URL они наиболее полезны, но имейте в виду, что encodeURIComponent() будет кодировать символы / в имени пути, которые вы, по всей видимости, не хотите кодировать. К тому же она будет преобразовывать пробелы внутри параметра запроса в %20, хотя в указанной порции URL пробелы должны кодироваться с помощью +.

Фундаментальная проблема со всеми унаследованными функциями заключается в том, что они стремятся применить ко всем частям URL единственную схему кодирования, хотя разные порции URL используют разные кодирования.

Если вас интересует надлежащим образом сформатированный и закодированный URL, то решением будет применение класса URL для всех выполняемых манипуляций с URL.

11.10. Таймеры

С первых дней появления JavaScript в веб-браузерах были определены две функции, `setTimeout()` и `setInterval()`, которые позволяют программам запрашивать у браузера вызов функции по истечении указанного времени или повторяющийся вызов функции через заданные интервалы. Такие функции никогда не были стандартизированы как часть базового языка, но они работают во всех браузерах и в Node, де-факто став частью стандартной библиотеки JavaScript.

В первом аргументе `setTimeout()` передается функция, а во втором — число, которое указывает, сколько миллисекунд должно пройти, прежде чем функция будет вызвана. По истечении заданного времени (и возможно чуть позже, если система занята) функция вызывается без аргументов. Например, ниже показаны три вызова `setTimeout()`, которые выводят консольные сообщения через одну, две и три секунды:

```
setTimeout(() => { console.log("Ready..."); }, 1000);
setTimeout(() => { console.log("set..."); }, 2000);
setTimeout(() => { console.log("go!"); }, 3000);
```

Обратите внимание, что функция `setTimeout()` не ожидает истечения указанного времени до того, как возвратить управление. Все три строки кода, приведенные выше, выполняются почти мгновенно, но затем ничего не происходит, пока не пройдет 1000 миллисекунд.

Когда второй аргумент `setTimeout()` опущен, по умолчанию он принимается равным 0. Тем не менее, это вовсе не означает, что указанная в первом аргументе функция будет вызвана немедленно. Взамен функция регистрируется для вызова "как можно быстрее". Если браузер чрезвычайно занят обработкой пользовательского ввода или других событий, то может пройти 10 и более миллисекунд, прежде чем функция вызовется.

Функция `setTimeout()` регистрирует функцию для однократного вызова. Иногда зарегистрированная функция сама будет вызывать `setTimeout()`, чтобы запланировать еще один вызов в будущем. Однако если вы хотите вызывать функцию многократно, тогда часто проще использовать `setInterval()`. Функция `setInterval()` принимает такие же два аргумента, как `setTimeout()`, но вызывает функцию каждый раз, когда истекает (приблизительно) указанное количество миллисекунд.

Функции `setTimeout()` и `setInterval()` возвращают значение. Если вы сохраните возвращенное значение в переменной, то затем сможете задействовать его позже для отмены выполнения зарегистрированной функции, передав переменную `clearTimeout()` или `clearInterval()`. Возвращаемым значением обычно будет число в веб-браузерах и объект в Node. Фактический тип не

играет роли, и вы должны обращаться с ним как с непрозрачным значением. Единственное, что можно делать с этим значением — передавать его функции clearTimeout() для отмены выполнения той функции, которая была зарегистрирована посредством setTimeout() (предполагая, что она еще не вызывалась), или для прекращения повторяющегося выполнения функции, зарегистрированной с помощью setInterval().

Далее представлен пример, в котором демонстрируется применение функций setTimeout(), setInterval() и clearInterval() для отображения простых цифровых часов с использованием API-интерфейса Console:

```
// Раз в секунду: очистить консоль и вывести текущее время.
let clock = setInterval(() => {
    console.clear();
    console.log(new Date().toLocaleTimeString());
}, 1000);

// Спустя 10 секунд: прекратить повторение выполнения кода выше.
setTimeout(() => { clearInterval(clock); }, 10000);
```

Вы снова увидите функции setTimeout() и setInterval() в действии, когда будет обсуждаться асинхронное программирование в главе 13.

11.11. Резюме

Изучение языка программирования — это не только овладение грамматикой. Не менее важно исследовать стандартную библиотеку, чтобы ознакомиться со всеми инструментами, которые сопровождают язык. Ниже перечислены основные моменты, касающиеся стандартной библиотеки, которые рассматривались в главе.

- Важные структуры данных, такие как Set, Map и типизированные массивы.
- Классы Date и URL для работы со значениями даты и указателями URL.
- Грамматика регулярных выражений JavaScript и класс RegExp для сопоставления с текстовым образцом.
- Библиотека интернационализации JavaScript для форматирования значений даты, времени и чисел, а также для сортировки строк.
- Объект JSON для сериализации и десериализации простых структур данных и объект console для вывода сообщений.

ГЛАВА 12

Итераторы и генераторы

Итерируемые объекты и ассоциированные с ними итераторы являются средством ES6, которое используется в книге повсеместно. Массивы (в том числе типизированные) итерируемы, в равной степени как строки и объекты Set и Map. Это означает возможность выполнения итерации — прохода — по содержимому таких структур данных с помощью цикла for/of, как было показано в подразделе 5.4.4:

```
let sum = 0;
for(let i of [1,2,3]) {      // Однократный проход по каждому значению
   sum += i;
}
sum    // => 6
```

Итераторы также можно применять с операцией ... для расширения или "распространения" итерируемого объекта в инициализаторе массива или в вызове функции, что демонстрировалось в подразделе 7.1.2:

```
let chars = [..."abcd"];    // chars == ["a", "b", "c", "d"]
let data = [1, 2, 3, 4, 5];
Math.max(...data)           // => 5
```

Итераторы можно использовать в деструктурирующем присваивании:

```
let purpleHaze = Uint8Array.of(255, 0, 255, 128);
let [r, g, b, a] = purpleHaze;          // a == 128
```

При выполнении итерации по объекту Map возвращаемыми значениями будут пары [ключ, значение], которые хорошо сочетаются с деструктурирующим присваиванием в цикле for/of:

```
let m = new Map([["one", 1], ["two", 2]]);
for(let [k,v] of m) console.log(k, v);  // Выводится 'one 1' и 'two 2'
```

Если вы хотите организовать итерацию только по ключам или только по значениям, а не по их парам, тогда можете применять методы keys() и values():

```
[...m]           // => [["one", 1], ["two", 2]]: итерация по умолчанию
[...m.entries()] // => [["one", 1], ["two", 2]]: то же самое
                 // обеспечивает метод entries()
[...m.keys()]    // => ["one", "two"]: метод keys() выполняет
                 // итерацию только по ключам в Map
[...m.values()]  // => [1, 2]: метод values() выполняет итерацию
                 // только по значениям в Map
```

Наконец, ряд встроенных функций и конструкторов, которые обычно используются с объектами `Array`, в действительности реализованы (в ES6 и последующих версиях) для приема произвольных итераторов. Конструктор `Set()` — один из API-интерфейсов такого рода:

```
// Строки итерируемы, поэтому два множества одинаковы:
new Set("abc")     // => new Set(["a", "b", "c"])
```

В настоящей главе объясняется работа итераторов и методика создания собственных структур данных, которые являются итерируемыми. После обсуждения базовых итераторов в главе раскрываются генераторы — мощное новое средство ES6, которое главным образом применяется в качестве очень легкого способа создания итераторов.

12.1. Особенности работы итераторов

Цикл `for/of` и операция распространения без проблем работают с итерируемыми объектами, но немаловажно понимать, что на самом деле должно произойти для того, чтобы заставить итерацию работать. Существуют три отдельных типа, которые необходимо освоить для понимания итерации в JavaScript. Во-первых, есть *итерируемые* объекты: типы наподобие `Array`, `Set` и `Map`, по которым можно организовать итерацию. Во-вторых, имеется сам объект *итератора*, который выполняет итерацию. И, в-третьих, есть объект *результата итерации*, который хранит результат каждого шага итерации.

Итерируемый объект — это любой объект со специальным итераторным методом, который возвращает объект итератора. Итератор — это любой объект с методом `next()`, который возвращает объект результата итерации. А объект *результата итерации* — это объект со свойствами по имени `value` и `done`. Для выполнения итерации по итерируемому объекту вы сначала вызываете его итераторный метод, чтобы получить объект итератора. Затем вы многократно вызываете метод `next()` объекта итератора до тех пор, пока свойство `done` возвращенного значения не окажется установленным в `true`. Сложность здесь в том, что итераторный метод итерируемого объекта не имеет обыкновенного имени, а взамен использует символьное имя `Symbol.iterator`. Таким образом, простой цикл `for/of` по итерируемому объекту `iterable` может быть записан в сложной форме:

```
let iterable = [99];
let iterator = iterable[Symbol.iterator]();
for(let result = iterator.next(); !result.done; result = iterator.next()) {
   console.log(result.value)  // result.value == 99
}
```

Объект итератора встроенных итерируемых типов данных сам является итерируемым. (То есть он имеет метод с символьным именем `Symbol.iterator`, который просто возвращает сам объект.) Иногда это полезно в коде следующего вида, когда вы хотите выполнить проход по "частично использованному" итератору:

```
let list = [1,2,3,4,5];
let iter = list[Symbol.iterator]();
let head = iter.next().value;       // head == 1
let tail = [...iter];               // tail == [2,3,4,5]
```

12.2. Реализация итерируемых объектов

Итерируемые объекты настолько полезны в ES6, что вы должны подумать о том, чтобы сделать собственные типы данных итерируемыми всякий раз, когда они представляют что-нибудь, допускающее итерацию. Классы Range, показанные в примерах 9.2 и 9.3 из главы 9, были итерируемыми. Классы Range применяли генераторные функции, чтобы превратить себя в итерируемые классы. Генераторы будут описаны далее в главе, но сначала мы реализуем класс Range еще раз, сделав его итерируемым без помощи генератора.

Чтобы сделать класс итерируемым, потребуется реализовать метод с символьным именем `Symbol.iterator`, который должен возвращать объект итератора, имеющий метод `next()`. А метод `next()` обязан возвращать объект результата итерации, который имеет свойство `value` и/или булевское свойство `done`. В примере 12.1 реализован итерируемый класс Range и показано, как создавать итерируемый объект, объект итератора и объект результата итерации.

Пример 12.1. Итерируемый числовой класс Range

```
/*
 * Объект Range представляет диапазон чисел {x: from <= x <= to}.
 * В классе Range определен метод has() для проверки,
 * входит ли заданное число в диапазон.
 * Класс Range итерируемый и обеспечивает проход по всем целым
 * числам внутри диапазона.
 */
class Range {
  constructor (from, to) {
    this.from = from;
    this.to = to;
  }

  // Сделать класс Range работающим подобно множеству Set чисел.
  has(x) { return typeof x === "number" && this.from <= x && x <= this.to; }

  // Возвратить строковое представление диапазона, используя запись множества.
  toString() { return `{ x | ${this.from} ≤ x ≤ ${this.to} }`; }

  // Сделать класс Range итерируемым за счет возвращения объекта итератора.
  // Обратите внимание на то, что именем этого метода является
  // специальный символ, а не строка.
```

```
    [Symbol.iterator]() {
        // Каждый экземпляр итератора обязан проходить по диапазону
        // независимо от других. Таким образом, нам нужна переменная
        // состояния, чтобы отслеживать местоположение в итерации.
        // Мы начинаем с первого целого числа, которое больше или равно from.
        let next = Math.ceil(this.from);   // Это значение мы возвращаем
                                           // следующим.
        let last = this.to;        // Мы не возвращаем ничего, что больше этого.
        return {                   // Это объект итератора.
            // Именно данный метод next() делает это объектом итератора.
            // Он обязан возвращать объект результата итерации.
            next() {
                return (next <= last)   // Если пока не возвратили последнее
                    ? { value: next++ } // значение, возвратить следующее
                                        // значение и инкрементировать его,
                    : { done: true };   // в противном случае указать,
                                        // что все закончено.
            },
            // Для удобства мы делаем сам итератор итерируемым.
            [Symbol.iterator]() { return this; }
        };
    }
}
for(let x of new Range(1,10)) console.log(x); // Выводятся числа от 1 до 10
[...new Range(-2,2)]                          // => [-2, -1, 0, 1, 2]
```

В дополнение к превращению своих классов в итерируемые иногда весьма полезно определить функции, которые возвращают итерируемые значения. Рассмотрим такие итерируемые альтернативы методам `map()` и `filter()` массивов JavaScript:

```
// Возвращает итерируемый объект, который проходит по результату
// применения f() к каждому значению из исходного итерируемого объекта.
function map(iterable, f) {
    let iterator = iterable[Symbol.iterator]();
    return {    // Этот объект является и итератором, и итерируемым.
        [Symbol.iterator]() { return this; },
        next() {
            let v = iterator.next();
            if (v.done) {
                return v;
            } else {
                return { value: f(v.value) };
            }
        }
    };
}
//Отобразить диапазон целых чисел на их квадраты и преобразовать в массив.
[...map(new Range(1,4), x => x*x)]  // => [1, 4, 9, 16]
```

```
// Возвращает итерируемый объект, который фильтрует указанный
// итерируемый объект, проходя только по тем элементам,
// для которых предикат возвращает true.
function filter(iterable, predicate) {
   let iterator = iterable[Symbol.iterator]();
   return {    // Этот объект является и итератором, и итерируемым.
      [Symbol.iterator]() { return this; },
      next() {
         for(;;) {
            let v = iterator.next();
            if (v.done || predicate(v.value)) {
               return v;
            }
         }
      }
   };
}
// Отфильтровать диапазон, чтобы остались только четные числа.
[...filter(new Range(1,10), x => x % 2 === 0)]  // => [2,4,6,8,10]
```

Одна из ключевых особенностей итерируемых объектов заключается в том, что они в своей основе ленивые: если для получения следующего значения требуется вычисление, то вычисление может быть отложено до момента, когда значение фактически понадобится. Пусть, например, у вас есть очень длинная строка текста, которую вы хотите разбить на отдельные слова. Вы могли бы просто воспользоваться методом `split()` вашей строки, но в таком случае до того, как удастся задействовать даже первое слово, должна быть полностью обработана вся строка. И в итоге вы выделили бы много памяти для возвращенного массива и всех строк внутри него. Ниже приведен код функции, которая позволяет ленивым образом проходить по словам в строке, не сохраняя сразу их всех в памяти (в ES2020 эту функцию можно было бы реализовать гораздо легче с применением метода `matchAll()`, возвращающего итератор, который был описан в подразделе 11.3.2):

```
function words(s) {
   var r = /\s+|$/g;    // Соответствует одному или большему
                        // количеству пробелов или концу.
   r.lastIndex = s.match(/[^ ]/).index; // Начать сопоставление
              // с первого символа, отличающегося от пробела.
   return {    // Возвратить итерируемый объект итератора.
      [Symbol.iterator]() {          // Это делает итерируемый объект.
         return this;
      },
      next() {                       // Это делает объект итератора.
         let start = r.lastIndex;    // Продолжить там, где закончи-
                                     // лось последнее совпадение.
         if (start < s.length) {     // Если работа не закончена.
            let match = r.exec(s);   // Соответствует следующей
                                     // границе слова.
```

```
            if (match) {   // Если она найдена, то возвратить слово.
                return { value: s.substring(start, match.index) };
            }
        }
        return { done: true };      // В противном случае указать,
                                    // что работа закончена.
    }
  };
}
[...words(" abc def ghi! ")]        // => ["abc", "def", "ghi!"]
```

12.2.1. "Закрытие" итератора: метод `return()`

Представим себе вариант итератора `words()` на JavaScript стороны сервера, который в своем аргументе принимает не исходную строку, а имя файла, открывает этот файл, читает из него строки и проходит по словам в этих строках. В большинстве операционных систем программы, которые открывают файлы для чтения из них, должны не забывать о закрытии файлов, когда чтение завершено, так что наш гипотетический итератор обязан гарантировать закрытие файла после того, как метод `next()` возвратит последнее слово в нем.

Но итераторы не всегда работают до конца: цикл `for/of` может быть прекращен посредством `break` или `return` либо из-за исключения. Аналогично, когда итератор используется с деструктурирующей операцией, метод `next()` вызывается лишь столько раз, сколько достаточно для получения значений для всех указанных переменных. Итератор может иметь гораздо больше значений, которые он способен возвратить, но они никогда не будут запрошены.

Даже когда наш гипотетический итератор по словам в файле никогда не дорабатывает до конца, то ему все равно необходимо закрыть файл, который он открыл. По этой причине объекты итераторов могут реализовывать вместе с методом `next()` метод `return()`. Если итерация останавливается до того, как `next()` возвратил результат итерации со свойством `done`, установленным в `true` (чаще всего из-за преждевременного покидания цикла `for/of` через оператор `break`), тогда интерпретатор проверит, есть ли у объекта итератора метод `return()`. Если метод `return()` существует, то интерпретатор вызовет его без аргументов, давая итератору шанс закрыть файлы, освободить память и как-то иначе произвести очистку после себя. Метод `return()` обязан возвращать объект результата итерации. Свойства этого объекта игнорируются, но возвращение значения, отличающегося от объекта, приведет к ошибке.

Цикл `for/of` и операция распространения — действительно полезные средства JavaScript, так что при создании API-интерфейсов рекомендуется по возможности применять их. Но необходимость работать с итерируемым объектом, объектом его итератора и объектами результатов итераций несколько усложняет процесс. К счастью, как будет показано в остальных разделах главы, генераторы способны значительно упростить создание специальных итераторов.

12.3. Генераторы

Генератор — это своего рода итератор, определенный с помощью нового мощного синтаксиса ES6; он особенно удобен, когда значения, по которым нужно выполнять итерацию, являются не элементами какой-то структуры данных, а результатом вычисления.

Для создания генератора сначала потребуется определить *генераторную функцию*. Генераторная функция синтаксически похожа на обыкновенную функцию JavaScript, но определяется с помощью ключевого слова `function*` вместо `function`. (Формально это не новое ключевое слово, а просто символ * после ключевого слова `function` и перед именем функции.) При вызове генераторной функции тело функции фактически не выполняется, но взамен возвращается объект генератора, который является итератором. Вызов его метода `next()` заставляет тело генераторной функции выполняться с самого начала (или с любой текущей позиции), пока не будет достигнут оператор `yield`. Оператор `yield` появился в ES6 и кое в чем похож на оператор `return`. Значение оператора `yield` становится значением, которое возвращается вызовом метода `next()` итератора. Вот пример, который должен все прояснить:

```
// Генераторная функция, которая выдает набор простых чисел
// с одной цифрой (с основанием 10).
function* oneDigitPrimes() {    // При вызове этой функции код
                                // не выполняется, а просто
   yield 2;                     // возвращается объект генератора.
                                // Вызов метода next()
   yield 3;                     // данного генератора приводит
                                // к выполнению кода до тех пор,
   yield 5;                     // пока оператор yield не предоставит
                                // возвращаемое значение
   yield 7;                     // для метода next().
}

// Когда мы вызываем генераторную функцию, то получаем генератор.
let primes = oneDigitPrimes();

// Генератор - это объект итератора, который проходит
// по выдаваемым значениям.
primes.next().value             // => 2
primes.next().value             // => 3
primes.next().value             // => 5
primes.next().value             // => 7
primes.next().done              // => true

// Генераторы имеют метод Symbol.iterator, что делает их итерируемыми.
primes[Symbol.iterator]()       // => primes

// Мы можем использовать генераторы подобно другим итерируемым типам.
[...oneDigitPrimes()]           // => [2,3,5,7]
let sum = 0;
for(let prime of oneDigitPrimes()) sum += prime;
sum                             // => 17
```

В показанном выше примере для определения генератора мы использовали оператор function*. Однако подобно обыкновенным функциям мы можем определять генераторы также в форме выражений. Нужно лишь поместить звездочку после ключевого слова function:

```
const seq = function*(from,to) {
   for(let i = from; i <= to; i++) yield i;
};
[...seq(3,5)]  // => [3, 4, 5]
```

В классах и объектных литералах можно применять сокращенную запись, полностью опуская ключевое слово function при определении методов. Чтобы определить генератор в таком контексте, мы просто используем звездочку перед именем метода, где находилось бы ключевое слово function, если бы оно было указано:

```
let o = {
   x: 1, y: 2, z: 3,
   // Генератор, который выдает каждый ключ этого объекта.
   *g() {
      for(let key of Object.keys(this)) {
         yield key;
      }
   }
};
[...o.g()]  // => ["x", "y", "z", "g"]
```

Обратите внимание, что записать генераторную функцию с применением синтаксиса стрелочных функций не удастся.

Генераторы часто упрощают определение итерируемых классов. Мы можем заменить метод [Symbol.iterator](), представленный в примере 12.1, намного более короткой генераторной функцией *[Symbol.iterator](), которая выглядит следующим образом:

```
*[Symbol.iterator]() {
   for(let x = Math.ceil(this.from); x <= this.to; x++) yield x;
}
```

Чтобы увидеть такую основанную на генераторе функцию итератора в контексте, просмотрите пример 9.3 в главе 9.

12.3.1. Примеры генераторов

Генераторы более интересны, если они действительно *генерируют* выдаваемые значения, выполняя вычисление какого-нибудь вида. Скажем, ниже приведена генераторная функция, которая выдает числа Фибоначчи:

```
function* fibonacciSequence() {
   let x = 0, y = 1;
   for(;;) {
      yield y;
      [x, y] = [y, x+y];  // Примечание: деструктурирующее присваивание.
   }
}
```

Обратите внимание, что генераторная функция `fibonacciSequence()` содержит бесконечный цикл и постоянно выдает значения без возврата. В случае использования этого генератора с операцией распространения `...` цикл будет выполняться до тех пор, пока не исчерпается память и произойдет аварийный отказ программы. Тем не менее, соблюдая осторожность, данный генератор можно применять в цикле `for/of`:

```
// Возвращает n-тое число Фибоначчи.
function fibonacci(n) {
   for(let f of fibonacciSequence()) {
      if (n-- <= 0) return f;
   }
}
fibonacci(20)                // => 10946
```

Бесконечный генератор подобного рода становится более полезным, когда используется в генераторе `take()` следующего вида:

```
// Выдает первые n элементов указанного итерируемого объекта.
function* take(n, iterable) {
  let it = iterable[Symbol.iterator](); // Получить итератор для
                                        // итерируемого объекта.
  while(n-- > 0) {           // Цикл n раз:
    let next = it.next();    // Получить следующий элемент из итератора.
    if (next.done) return;   // Если больше нет значений, тогда
                             // выполнить возврат раньше,
    else yield next.value; // иначе выдать значение.
  }
}

// Массив из первых пяти чисел Фибоначчи.
[...take(5, fibonacciSequence())]    // => [1, 1, 2, 3, 5]
```

Вот еще одна полезная генераторная функция, которая чередует элементы множества итерируемых объектов:

```
// Для заданного массива итерируемых объектов выдает
// их элементы в чередующемся порядке.
function* zip(...iterables) {
   // Получить итератор для каждого итерируемого объекта.
   let iterators = iterables.map(i => i[Symbol.iterator]());
   let index = 0;
   while(iterators.length > 0) {       // Пока есть какие-то итераторы.
      if (index >= iterators.length) {// Если мы достигли последнего
                                      // итератора, тогда вернуться
         index = 0;                   // к первому итератору.
      }
      let item = iterators[index].next(); //Получить следующий элемент
                                  // от следующего итератора.
      if (item.done) {            // Если этот итератор закончил работу,
         iterators.splice(index, 1); // тогда удалить его из массива.
      }
      else {                      // Иначе
```

```
            yield item.value;      // выдать текущее значение
            index++;               // и перейти к следующему итератору.
        }
    }
}
// Чередовать элементы трех итерируемых объектов.
[...zip(oneDigitPrimes(),"ab",[0])]        // => [2,"a",0,3,"b",5,7]
```

12.3.2. yield* и рекурсивные генераторы

Помимо генератора `zip()`, определенного в предыдущем примере, полезно иметь аналогичную генераторную функцию, которая выдает элементы множества итерируемых объектов последовательно, не чередуя их. Мы могли бы написать такой генератор следующим образом:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        for(let item of iterable) {
            yield item;
        }
    }
}
[...sequence("abc",oneDigitPrimes())] // => ["a","b","c",2,3,5,7]
```

Процесс выдачи элементов какого-то другого итерируемого объекта достаточно распространен в генераторных функциях, поэтому в ES6 для него предусмотрен специальный синтаксис. Ключевое слово `yield*` похоже на `yield`, но вместо выдачи одиночного значения оно проходит по итерируемому объекту и выдает каждое результирующее значение. Генераторную функцию `sequence()`, которую мы применяли ранее, за счет использования `yield*` можно упростить:

```
function* sequence(...iterables) {
    for(let iterable of iterables) {
        yield* iterable;
    }
}
[...sequence("abc",oneDigitPrimes())] // => ["a","b","c",2,3,5,7]
```

Метод `forEach()` массива часто оказывается элегантным способом прохода в цикле по элементам массива, поэтому у вас может возникнуть соблазн реализовать функцию `sequence()`, как показано ниже:

```
function* sequence(...iterables) {
    iterables.forEach(iterable => yield* iterable ); // Ошибка!
}
```

Однако такой код работать не будет. Ключевые слова `yield` и `yield*` могут применяться только внутри генераторных функций, но вложенная стрелочная функция в данном коде является обыкновенной, а не генераторной функцией `function*`, так что использование `yield` в ней не разрешено.

Ключевое слово yield* можно применять с итерируемым объектом любого вида, включая итерируемые объекты, которые реализованы с помощью генераторов. Таким образом, ключевое слово yield* позволяет определять рекурсивные генераторы, и вы можете использовать это средство, например, чтобы обеспечить простой нерекурсивный обход рекурсивно определенной древовидной структуры.

12.4. Расширенные возможности генераторов

Генераторные функции чаще всего применяются для создания итераторов, но фундаментальная особенность генераторов заключается в том, что они позволяют приостанавливать вычисление, выдавать промежуточные результаты и позже возобновлять вычисление. Это означает, что генераторы обладают возможностями, выходящими за рамки итераторов, и мы исследуем их в последующих разделах.

12.4.1. Возвращаемое значение генераторной функции

Генераторные функции, встречавшиеся до сих пор, не имели операторов return, но даже если имели, то они использовались для преждевременного возврата, а не для того, что возвратить значение. Тем не менее, как и любая функция, генераторная функция способна возвращать значение. Для понимания того, что происходит в этом случае, вспомните, каким образом работает итерация. Возвращаемым значением функции next() является объект, который имеет свойство value и/или свойство done. В типичных итераторах и генераторах, если определено свойство value, то свойство done не определено или равно false. Если свойство done равно true, тогда свойство value не определено. Но в ситуации с генератором, который возвращает значение, финальный вызов next возвращает объект, в котором определены оба свойства, value и done. Свойство value хранит возвращаемое значение генераторной функции, а свойство done равно true, указывая на то, что больше нет значения для итерации. Цикл for/of и операция распространения игнорируют это финальное значение, но оно доступно коду, в котором производится итерация вручную с помощью явных вызовов next():

```
function *oneAndDone() {
   yield 1;
   return "done";
}

// При нормальной итерации возвращаемое значение не появляется.
[...oneAndDone()]           // => [1]
// Но оно доступно в случае явного вызова next().
let generator = oneAndDone();
generator.next()            // => { value: 1, done: false}
generator.next()            // => { value: "done", done: true }
// Если генератор уже закончил работу, то возвращаемое значение
// больше не возвращается.
generator.next()            // => { value: undefined, done: true }
```

12.4.2. Значение выражения yield

В предыдущем обсуждении мы обходились с yield как с оператором, который принимает значение, но не имеет собственного значения. Однако на самом деле yield является выражением и может иметь значение.

Когда вызывается метод next() генератора, генераторная функция выполняется до тех пор, пока не достигает выражения yield. Выражение, указанное после ключевого слова yield, вычисляется и его значение становится возвращаемым значением вызова next(). В данной точке генераторная функция останавливает выполнение прямо на середине вычисления выражения yield. Когда метод next() генератора вызывается в следующий раз, переданный next() аргумент становится значением выражения yield, вычисление которого было приостановлено. Таким образом, генератор возвращает значения вызывающему коду с помощью yield, а вызывающий код передает значения генератору посредством next(). Генератор и вызывающий код — два отдельных потока выполнения, передающие значения (и управление) туда и обратно. Следующий код иллюстрирует сказанное:

```
function* smallNumbers() {
    console.log("next() вызывается первый раз; аргумент отбрасывается");
    let y1 = yield 1;    // y1 == "b"
    console.log("next() вызывается второй раз с аргументом", y1);
    let y2 = yield 2;    // y2 == "c"
    console.log("next() вызывается третий раз с аргументом", y2);
    let y3 = yield 3;    // y3 == "d"
    console.log("next() вызывается четвертый раз с аргументом", y3);
    return 4;
}
let g = smallNumbers();
console.log("генератор создан; никакой код пока не выполнялся");
let n1 = g.next("a");   // n1.value == 1
console.log("генератор выдает", n1.value);
let n2 = g.next("b");   // n2.value == 2
console.log("генератор выдает", n2.value);
let n3 = g.next("c");   // n3.value == 3
console.log("генератор выдает", n3.value);
let n4 = g.next("d");   // n4 == { value: 4, done: true }
console.log("генератор возвращает", n4.value);
```

Во время выполнения код производит показанный ниже вывод, который демонстрирует обмен значениями между двумя блоками кода:

```
генератор создан; никакой код пока не выполнялся
next() вызывается первый раз; аргумент отбрасывается
генератор выдает 1
next() вызывается второй раз с аргументом b
генератор выдает 2
next() вызывается третий раз с аргументом c
генератор выдает 3
next() вызывается четвертый раз с аргументом d
генератор возвращает 4
```

Обратите внимание на асимметрию в этом коде. Первый вызов `next()` запускает генератор, но значение, переданное данному вызову, оказывается недоступным генератору.

12.4.3. Методы `return()` и `throw()` генератора

Вы уже видели, что можно получать значения, выдаваемые или возвращаемые генераторной функцией. Кроме того, можно передавать значения выполняющемуся генератору, указывая их при вызове метода `next()` генератора.

Помимо предоставления входных данных генератору с помощью `next()` можно также изменять поток управления внутри генератора, вызывая его методы `return()` и `throw()`. Как должно быть понятно по их именам, вызов упомянутых методов генератора заставляет его возвратить значение или сгенерировать исключение, как если бы следующим оператором был `return` или `throw`.

Как объяснялось ранее в главе, если итератор определяет метод `return()` и итерация останавливается преждевременно, тогда интерпретатор автоматически вызывает метод `return()`, чтобы дать итератору возможность закрыть файлы или выполнить другую очистку. В случае генераторов нельзя определить специальный метод `return()` для поддержки очистки, но можно структурировать код генератора так, чтобы задействовать оператор `try/finally`, который гарантирует выполнение необходимой очистки (в блоке `finally`), когда происходит возврат из генератора. За счет принудительного возврата из генератора его встроенный метод `return()` обеспечивает выполнение кода очистки, когда генератор больше не будет использоваться.

Подобно тому, как метод `next()` генератора позволяет передавать произвольные значения выполняющемуся генератору, метод `throw()` генератора предоставляет нам способ отправки генератору произвольных сигналов (в форме исключений). Вызов метода `throw()` всегда инициирует исключение внутри генератора. Но если генераторная функция написана с соответствующим кодом обработки исключений, то исключение не обязано быть фатальным, а взамен может служить средством изменения поведения генератора. Представьте себе, скажем, генератор счетчика, который выдает строго возрастающую последовательность целых чисел. Его можно было бы реализовать так, чтобы исключение, отправленное с помощью `throw()`, сбрасывало счетчик в ноль.

Когда генератор применяет `yield*` для выдачи значений из какого-то другого итерируемого объекта, то вызов метода `next()` генератора приводит к вызову метода `next()` итерируемого объекта. То же самое справедливо в отношении методов `return()` и `throw()`. Если генератор использует `yield*` с итерируемым объектом, в котором эти методы определены, тогда вызов метода `return()` или `throw()` генератора приводит к вызову метода `return()` или `throw()` итератора по очереди. Все итераторы *обязаны* иметь метод `next()`. Итераторы, которые нуждаются в очистке после незавершенной итерации, *должны* определять метод `return()`. И любой итератор *может* определять метод `throw()`, хотя я не знаю никаких практических причин для этого.

12.4.4. Финальное замечание о генераторах

Генераторы — очень мощная обобщенная структура управления. Они дают нам возможность приостанавливать вычисление с помощью `yield` и снова перезапускать его в произвольно более позднее время с произвольным входным значением. Генераторы можно применять для создания своего рода кооперативной потоковой системы внутри однопоточного кода JavaScript. Кроме того, генераторы можно использовать для маскировки асинхронных частей программы, чтобы код выглядел последовательным и синхронным, даже если некоторые вызовы функций на самом деле являются асинхронными и зависят от событий из сети.

Попытки делать такие вещи посредством генераторов приводят к появлению кода, который невероятно сложно понять или объяснить. Тем не менее, это делалось, и единственным действительно практичным сценарием применения было управление асинхронным кодом. Однако теперь в JavaScript есть ключевые слова `async` и `await` (см. главу 13), предназначенные именно для такой цели, и больше нет никаких причин злоупотреблять генераторами подобным образом.

12.5. Резюме

Ниже перечислены основные моменты, которые рассматривались в главе.

- Цикл `for/of` и операция распространения `...` работают с итерируемыми объектами.
- Объект является итерируемым, если он имеет метод с символьным именем `[Symbol.iterator]`, который возвращает объект итератора.
- Объект итератора имеет метод `next()`, который возвращает объект результата итерации.
- Объект результата итерации имеет свойство `value`, которое хранит следующее проходимое значение при его наличии. Если итерация закончилась, тогда объект результата должен иметь свойство `done`, установленное в `true`.
- Вы можете реализовывать собственные итерируемые объекты, определяя метод `[Symbol.iterator]()`, который возвращает объект с методом `next()`, возвращающим объекты результатов итерации. Вы также можете реализовывать функции, которые принимают аргументы итераторов и возвращают значения итераторов.
- Генераторные функции (функции, определенные с помощью `function*` вместо `function`) представляют собой еще один способ определения итераторов.
- Когда вы вызываете генераторную функцию, ее тело выполняется не сразу; взамен возвращается значение, которое является объектом итератора. Каждый раз, когда вызывается метод `next()` этого итератора, выполняется еще одна порция генераторной функции.
- В генераторных функциях можно использовать ключевое слово `yield` для указания значений, которые возвращаются итератором. Каждый вызов `next()` приводит к тому, что генераторная функция выполняется до следующего выражения `yield`. Затем значение выражения `yield` становится значением, возвращаемым итератором. Когда выражений `yield` больше нет, тогда происходит возврат из генераторной функции и итерация завершается.

ГЛАВА 13

Асинхронный JavaScript

Некоторые компьютерные программы, подобные реализациям научного моделирования и моделей машинного обучения, привязаны к вычислениям: они выполняются непрерывно, без пауз, пока не вычислят свои результаты. Однако большинство реальных компьютерных программ являются в значительной степени *асинхронными*. Другими словами, им часто приходится останавливать вычисление в ожидании поступления данных или возникновения какого-то события. Программы JavaScript в веб-браузере обычно *управляются событиями*, т.е. они ждут, когда пользователь щелкнет или коснется, прежде чем фактически что-то делать. И серверы, основанные на JavaScript, как правило, ожидают поступления по сети клиентских запросов, прежде чем делать что-нибудь.

Такой вид асинхронного программирования — общеизвестное явление в JavaScript, и в главе описаны три важных языковых средства, которые помогают облегчить работу с асинхронным кодом. Объекты `Promise`, появившиеся в ES6, представляют пока не доступный результат асинхронной операции. Ключевые слова `async` и `await` были введены в ES2017 и предлагают новый синтаксис, который упрощает асинхронное программирование, позволяя структурировать основанный на `Promise` код, как если бы он был синхронным. Наконец, асинхронные итераторы и цикл `for/await` появились в ES2018 и дают нам возможность работать с потоками асинхронных событий, используя простые циклы, которые выглядят синхронными.

По иронии судьбы, хотя JavaScript предоставляет мощные средства для работы с асинхронным кодом, в базовом языке отсутствуют возможности, которые сами были бы асинхронными. Следовательно, для демонстрации объектов `Promise`, `async`, `await` и `for/await` мы сначала обратимся к JavaScript на сторонах клиента и сервера, чтобы объяснить ряд асинхронных возможностей веб-браузеров и Node. (Вы получите больше сведений о JavaScript на сторонах клиента и сервера в главах 15 и 16.)

13.1. Асинхронное программирование с использованием обратных вызовов

На самом фундаментальном уровне асинхронное программирование на JavaScript производится с помощью *обратных вызовов*. Обратный вызов — это функция, которую вы пишете и затем передаете какой-то другой функции. Затем другая функция вызывает вашу функцию ("делает обратный вызов"), когда удовлетворяется определенное условие или происходит некоторое (асинхронное) событие. Вызов предоставленной вами функции обратного вызова уведомляет вас об условии или событии, а иногда вызов будет включать аргументы функции, которые обеспечивают дополнительные детали. Сказанное легче понять на конкретных примерах, и в последующих подразделах демонстрируются разнообразные формы асинхронного программирования, основанного на обратных вызовах, с применением JavaScript стороны клиента и Node.

13.1.1. Таймеры

Один из простейших видов асинхронности касается ситуации, когда вам необходимо запустить какой-то код по истечении определенного времени. Как было показано в разделе 11.10, для этого используется функция `setTimeout()`:

```
setTimeout(checkForUpdates, 60000);
```

В первом аргументе `setTimeout()` передается функция, а во втором — временной интервал в миллисекундах. В предыдущей строке кода гипотетическая функция `checkForUpdates()` будет вызвана через 60 000 миллисекунд (1 минуту) после вызова `setTimeout()`. Кроме того, `checkForUpdates()` является функцией обратного вызова, которая может быть определена в вашей программе, а `setTimeout()` — функцией, которую вы вызываете для регистрации функции обратного вызова и указания, при каких асинхронных условиях она должна вызываться.

Функция `setTimeout()` вызывает указанную функцию обратного вызова один раз, не передавая никаких аргументов, и затем забывает о ней. Если вы реализуете функцию, которая осуществляет проверку на предмет обновлений, тогда наверняка захотите запускать ее многократно. В таком случае вместо `setTimeout()` применяется `setInterval()`:

```
// Вызвать checkForUpdates через одну минуту и после этого
// снова каждую минуту.
let updateIntervalId = setInterval(checkForUpdates, 60000);

// setInterval() возвращает значение, которое можно использовать
// для останова повторяющихся вызовов, передав его clearInterval().
// (Аналогично setTimeout() возвращает значение,
// которое можно передать clearTimeout().)
function stopCheckingForUpdates() {
    clearInterval(updateIntervalId);
}
```

13.1.2. События

Программы JavaScript стороны клиента почти всегда управляются событиями: вместо того, чтобы выполнять какие-то заранее определенные вычисления, они обычно ожидают, пока пользователь что-то сделает, после чего реагируют на действия пользователя. Когда пользователь нажимает клавишу на клавиатуре, перемещает указатель мыши, щелкает кнопкой мыши или касается сенсорного экрана, веб-браузер генерирует *событие*. Программа JavaScript, управляемая событиями, регистрирует функции обратного вызова для указанных типов событий в заданных контекстах, а веб-браузер вызывает такие функции всякий раз, когда происходит указанные события. Эти функции обратного вызова называются *обработчиками событий* или *прослушивателями событий* и регистрируются с помощью `addEventListener()`:

```
// Запросить у веб-браузера возврат объекта, представляющего HTML-
// элемент <button>, который соответствует этому селектору CSS.
let okay = document.querySelector('#confirmUpdateDialog button.okay');

// Зарегистрировать функцию обратного вызова, которая должна
// вызываться, когда пользователь щелкает на данной кнопке.
okay.addEventListener('click', applyUpdate);
```

В приведенном примере `applyUpdate()` — гипотетическая функция обратного вызова, которая по нашему предположению реализована где-то в другом месте. Вызов `document.querySelector()` возвращает объект, который представляет указанный одиночный элемент веб-страницы. Для регистрации обратного вызова мы вызываем `addEventListener()` на этом элементе. В первом аргументе `addEventListener()` передается строка, указывающая вид интересующего события — в данном случае щелчок кнопкой мыши или касание сенсорного экрана. Если пользователь щелкнет на таком специфическом элементе веб-страницы или коснется его, тогда браузер вызовет нашу функцию обратного вызова `applyUpdate()`, передав ей объект, который включает детали о событии (наподобие момента времени и координат указателя мыши).

13.1.3. События сети

Еще одним распространенным источником асинхронности при программировании на JavaScript являются сетевые запросы. Программа JavaScript, выполняющаяся в браузере, может получать данные от веб-сервера посредством кода следующего вида:

```
function getCurrentVersionNumber(versionCallback) { // Обратите внимание
                                                    // на аргумент.
  // Сделать HTTP-запрос к API-интерфейсу версии сервера.
  let request = new XMLHttpRequest();
  request.open("GET", "http://www.example.com/api/version");
  request.send();

  // Зарегистрировать обратный вызов, который будет вызываться
  // при поступлении ответа.
  request.onload = function() {
```

```
        if (request.status === 200) {
            // Если состояние HTTP нормально, тогда получить
            // номер версии и вызвать обратный вызов.
            let currentVersion = parseFloat(request.responseText);
            versionCallback(null, currentVersion);
        } else {
            // В противном случае сообщить обратному вызову об ошибке.
            versionCallback(response.statusText, null);
        }
    };
    // Зарегистрировать еще один обратный вызов, который будет
    // вызываться при возникновении ошибок сети.
    request.onerror = request.ontimeout = function(e) {
        versionCallback(e.type, null);
    };
}
```

В коде JavaScript стороны клиента можно использовать класс `XMLHttpRequest` плюс функции обратного вызова, чтобы делать HTTP-запросы и асинхронным образом обрабатывать ответы сервера, когда они поступают[1]. Определенная здесь функция `getCurrentVersionNumber()` (мы можем представить себе, что она применяется в гипотетической функции `checkForUpdates()`, обсуждаемой в подразделе 13.1.1) делает HTTP-запрос и определяет обработчики событий, которые будут вызываться, когда получен ответ от сервера или запрос терпит неудачу из-за тайм-аута либо другой ошибки.

Обратите внимание, что в приведенном выше примере кода не вызывается функция `addEventListener()`, как делалось в предыдущем примере. Для большинства API-интерфейсов веб-браузеров (включая данный) обработчики событий можно определять вызовом `addEventListener()` на объекте, генерирующем события, и передавать имя интересующего события вместе с функцией обратного вызова. Тем не менее, регистрировать одиночный прослушиватель событий обычно также можно путем его присваивания прямо свойству объекта. Именно это мы делаем в примере кода, присваивая функции свойствам `onload`, `onerror` и `ontimeout`. По соглашению свойства прослушивателей событий такого рода всегда имеют имена, начинающиеся на *on*. Использование `addEventListener()` — более гибкий прием, поскольку он позволяет добавлять множество обработчиков событий. Но в случаях, когда вы уверены, что никакой другой код не будет нуждаться в регистрации прослушивателя для того же самого объекта и типа события, можете просто устанавливать соответствующее свойство в свой обратный вызов.

Еще один момент, который следует отметить о функции `getCurrentVersionNumber()` в показанном примере кода, заключается в том, что из-за выполнения асинхронного запроса она не может синхронно возвращать значение (текущий номер версии), в котором заинтересован вызывающий код. Взамен вызывающий

[1] Класс `XMLHttpRequest` не имеет ничего общего с XML. В современном JavaScript стороны клиента он почти полностью заменен API-интерфейсом `fetch()`, который раскрывается в подразделе 15.11.1. Показанный здесь пример кода — это последний пример на основе `XMLHttpRequest`, оставшийся в книге.

код передает функцию обратного вызова, которая вызывается, когда готов результат или произошла ошибка. В данном случае вызывающий код предоставляет функцию обратного вызова, которая ожидает двух аргументов. Если экземпляр XMLHttpRequest работает корректно, тогда getCurrentVersionNumber() вызывает обратный вызов, передавая null в первом аргументе и номер версии во втором аргументе. Если же возникла ошибка, то getCurrentVersionNumber() вызывает обратный вызов с передачей деталей ошибки в первом аргументе и null во втором аргументе.

13.1.4. Обратные вызовы и события в Node

Среда JavaScript стороны сервера Node.js глубоко асинхронна и определяет многочисленные API-интерфейсы, в которых применяются обратные вызовы и события. Скажем, стандартный API-интерфейс для чтения содержимого файла является асинхронным и вызывает функцию обратного вызова, когда содержимое файла прочитано:

```
const fs = require("fs");   // Модуль fs содержит API-интерфейсы,
                            // связанные с файловой системой.
let options = { // Объект для хранения параметров для нашей программы.
   // Здесь задаются параметры по умолчанию.
};

//Прочитать конфигурационный файл, затем вызвать функцию обратного вызова
fs.readFile("config.json", "utf-8", (err, text) => {
   if (err) {
//Если возникла ошибка, тогда отобразить предупреждение, но продолжить
      console.warn("Could not read config file:", err);
                  // Не удалось прочитать конфигурационный файл
   } else {
      // В противном случае произвести разбор содержимого файла
      // и присвоить объекту параметров.
      Object.assign(options, JSON.parse(text));
   }

   // В любом случае теперь мы можем начать выполнение программы.
   startProgram(options);
});
```

Функция fs.readFile() из Node принимает в своем последнем аргументе обратный вызов с двумя параметрами. Она асинхронно читает указанный файл и затем вызывает обратный вызов. Если файл был прочитан успешно, тогда она передает содержимое файла во втором аргументе обратного вызова. Если возникла ошибка, то она передает объект ошибки в первом аргументе обратного вызова. В этом примере мы выразили обратный вызов в виде стрелочной функции, что является лаконичным и естественным синтаксисом для простой операции подобного рода.

В Node также определено несколько API-интерфейсов, основанных на событиях. В следующей далее функции показано, как делать HTTP-запрос содержимого URL в Node. Она имеет два уровня асинхронного кода, поддерживаемых прослу-

шивателями событий. Обратите внимание, что для регистрации прослушивателей событий в Node используется метод on(), а не addEventListener():

```
const https = require("https");

// Читает текстовое содержимое URL и асинхронно передает
// его обратному вызову.
function getText(url, callback) {
   // Запустить HTTP-запрос GET для URL.
   request = https.get(url);

   // Зарегистрировать функцию для обработки события ответа.
   request.on("response", response => {
      // Событие ответа означает, что были получены заголовки ответа.
      let httpStatus = response.statusCode;

      // Тело HTTP-ответа еще не было получено.
      // Поэтому мы регистрируем дополнительные обработчики событий,
      // подлежащие вызову, когда поступит тело HTTP-ответа.
      response.setEncoding("utf-8"); // Мы ожидаем текст Unicode,
      let body = "";                 // который будем здесь накапливать

      // Этот обработчик событий вызывается, когда готова порция тела.
      response.on("data", chunk => { body += chunk; });

      // Этот обработчик событий вызывается, когда ответ завершен.
      response.on("end", () => {
         if (httpStatus === 200) {// Если HTTP-ответ был корректным,
            callback(null, body);// тогда передать тело ответа
                                 // обратному вызову.
         } else {        // В противном случае передать объект ошибки.
            callback(httpStatus, null);
         }
      });
   });

   // Мы также регистрируем обработчик событий
   // для низкоуровневых ошибок сети.
   request.on("error", (err) => {
      callback(err, null);
   });
}
```

13.2. Объекты Promise

Теперь, когда вы ознакомились с примерами асинхронного программирования на основе обратных вызовов и событий в средах JavaScript сторон клиента и сервера, мы можем перейти к исследованию объектов Promise — средства базового языка, которое предназначено для упрощения асинхронного программирования.

Объект Promise представляет результат асинхронного вычисления. Результат может уже быть готовым или нет, и API-интерфейс Promise намеренно неконкретен в этом отношении: не существует способа синхронно получить значение объекта Promise; вы можете только предложить Promise вызвать какую-то

функцию обратного вызова, когда значение готово. Если вы определяете асинхронный API-интерфейс вроде функции `getText()` в предыдущем разделе, но хотите сделать его основанным на `Promise`, тогда опустите аргумент обратного вызова и взамен возвращайте объект `Promise`. Затем вызывающий код может регистрировать для такого объекта `Promise` один или большее количество обратных вызовов, которые будут вызываться, когда асинхронное вычисление закончится.

Итак, на самом простом уровне объекты `Promise` — это лишь другой способ работы с обратными вызовами. Однако их применение обеспечивает практические преимущества. Одна настоящая проблема с асинхронным программированием на основе обратных вызовов заключается в том, что нередко приходится сталкиваться с обратными вызовами внутри обратных вызовов внутри обратных вызовов, когда строки кода имеют настолько широкие отступы, что их трудно читать. Объекты `Promise` позволяют выражать вложенные обратные вызовы подобного рода в виде более линейной *цепочки объектов* `Promise`, которая легче для чтения и понимания.

Другая проблема с обратными вызовами связана с тем, что они затрудняют обработку ошибок. Если асинхронная функция (или асинхронно вызванный обратный вызов) генерирует исключение, то оно не имеет никакой возможности распространиться обратно к инициатору асинхронной операции. В этом заключается фундаментальный факт асинхронного программирования: оно нарушает обработку исключений. Альтернативный вариант предусматривает педантичное отслеживание и распространение ошибок с помощью аргументов и возвращаемых значений обратных вызовов, но поступать так утомительно и вдобавок сложно обеспечить правильную реализацию. Объекты `Promise` помогают здесь в том, что стандартизируют методику обработки ошибок и предлагают способ корректного распространения ошибок через цепочку объектов `Promise`.

Обратите внимание, что объекты `Promise` представляют будущие результаты одиночных асинхронных вычислений. Тем не менее, их нельзя использовать для представления повторяющихся асинхронных вычислений. Например, позже в главе мы напишем основанную на `Promise` альтернативную версию функции `setTimeout()`. Но мы не можем применить объекты `Promise` для замены `setInterval()`, поскольку эта функция многократно вызывает функцию обратного вызова, на что объекты `Promise` не рассчитаны. Аналогично мы могли бы использовать объект `Promise` вместо обработчика событий загрузки объекта `XMLHttpRequest`, т.к. данный обратный вызов вызывается только один раз. Но, как правило, мы не будем применять объект `Promise` на месте обработчика событий щелчка для объекта кнопки HTML, потому что обычно желательно предоставить пользователю возможность щелкать на кнопке много раз.

Вот что вы найдете в последующих подразделах:

- объяснение терминологии, связанной с объектами `Promise`, и обсуждение базового использования `Promise`;
- исследование способов выстраивания объектов `Promise` в цепочки;
- демонстрацию создания собственных API-интерфейсов на основе `Promise`.

Объекты Promise поначалу кажутся простыми, и базовый сценарий применения объектов Promise действительно прост и прямолинеен. Но они могут становиться удивительно запутанными для всего, что выходит за рамки простейших сценариев использования. Объекты Promise являются мощной идиомой для асинхронного программирования, но вы должны очень хорошо их понимать, чтобы применять корректно и уверенно. Однако вам стоит потратить время на развитие такого глубокого понимания, и я призываю вас внимательно изучить эту длинную главу.

13.2.1. Использование объектов Promise

С появлением объектов Promise в базовом языке JavaScript веб-браузеры начали внедрять API-интерфейсы, основанные на Promise. В предыдущем разделе мы реализовали функцию getText(), которая делала асинхронный HTTP-запрос и передавала тело HTTP-ответа указанной функции обратного вызова в виде строки. Представим себе вариант этой функции, getJSON(), который производит разбор тела HTTP-ответа как данные JSON и возвращает объект Promise вместо принятия аргумента обратного вызова. Мы реализуем функцию getJSON() позже в главе, а пока давайте посмотрим, как бы мы применяли такую служебную функцию, возвращающую Promise:

```
getJSON(url).then(jsonData => {
    // Это функция обратного вызова, которая будет вызываться
    // асинхронным образом с разобранным значением JSON,
    // когда оно становится доступным.
});
```

Функция getJSON() запускает асинхронный HTTP-запрос для указанного URL и затем, пока запрос остается ожидающим решения, возвращает объект Promise. Объект Promise определяет метод экземпляра then(). Взамен передачи нашей функции обратного вызова прямо в getJSON() мы передаем ее методу then(). Когда поступает HTTP-ответ, выполняется разбор его тела как данных JSON, а результирующее разобранное значение передается функции, которую мы передали then().

Вы можете считать метод then() методом регистрации обратного вызова, похожим на метод addEventListener(), который используется для регистрации обработчиков событий в JavaScript стороны клиента. Если вы вызовите метод then() объекта Promise несколько раз, то каждая указанная вами функция будет вызвана, когда обещанное вычисление завершится.

Тем не менее, в отличие от многих прослушивателей событий объект Promise представляет одиночное вычисление, и каждая функция, зарегистрированная посредством then(), будет вызываться только один раз. Стоит отметить, что функция, переданная вами then(), вызывается асинхронным образом, даже если при вызове then() асинхронное вычисление уже закончилось.

На простом синтаксическом уровне метод then() является отличительной особенностью объектов Promise, и идиоматично дополнять вызовом .then()

непосредственно вызов функции, которая возвращает Promise, без промежуточного шага присваивания объекта Promise какой-то переменной.

Также идиоматично именовать функции, возвращающие объекты Promise, и функции, потребляющие результаты объектов Promise, с помощью глаголов, что приводит к получению кода, особо легкого для чтения (на английском):

```
// Предположим, что есть функция вроде этой для отображения
// профиля пользователя.
function displayUserProfile(profile) { /* реализация опущена */ }

// Вот как можно было бы использовать эту функцию с объектом Promise.
// Обратите внимание, что данная строка кода читается
// почти как предложение на английском языке:
getJSON("/api/user/profile").then(displayUserProfile);
```

Обработка ошибок с помощью объектов Promise

Асинхронные операции, особенно связанные с работой в сети, обычно могут отказывать по нескольким причинам, и необходимо писать надежный код для обработки ошибок, которые неизбежно будут возникать.

Для объектов Promise мы можем делать это, передавая методу then() вторую функцию:

```
getJSON("/api/user/profile").then(displayUserProfile, handleProfileError);
```

Объект Promise представляет будущий результат асинхронного вычисления, которое происходит после создания объекта Promise. Поскольку вычисление выполняется после того, как нам возвращается объект Promise, вычисление не способно возвращать значение традиционным образом или генерировать исключение, которое мы могли бы перехватить. Функции, передаваемые then(), предоставляют альтернативы. Когда нормально завершается синхронное вычисление, оно просто возвращает результат в вызывающий код. Когда нормально завершается асинхронное вычисление, основанное на Promise, оно передает свой результат функции, которая является первым аргументом then().

Когда что-то идет не так в синхронном вычислении, оно генерирует исключение, которое распространяется вверх по стеку вызовов до тех пор, пока не встретится конструкция catch для его обработки. Когда выполняется асинхронное вычисление, вызывающий его код больше не находится в стеке, поэтому если что-то пошло не так, то попросту невозможно сгенерировать исключение с его распространением обратно в вызывающий код.

Взамен асинхронные вычисления на основе Promise передают исключение (обычно как объект Error какого-то вида, хотя это не обязательно) второй функции, переданной then(). Таким образом, если в показанном выше коде функция getJSON() выполняется нормально, тогда она передает свой результат displayUserProfile(). В случае возникновения ошибки (пользователь не вошел в систему, сервер не работает, подключение к Интернету пользователя разорвано, случился тайм-аут запроса и т.д.) функция getJSON() передает handleProfileError() объект Error.

На практике методу then() редко передаются две функции. Существует лучший и более идиоматичный способ обработки ошибок при работе с объектами Promise. Для его понимания сначала посмотрим, что происходит, если getJSON() завершается нормально, но в displayUserProfile() возникает ошибка. Эта функция обратного вызова асинхронно вызывается, когда происходит возврат из getJSON(), а потому она тоже асинхронна и не может осмысленно генерировать исключение (т.к. в стеке вызовов нет кода для его обработки).

Более идиоматичный способ обработки ошибок в данном коде выглядит следующим образом:

```
getJSON("/api/user/profile").then(displayUserProfile).
        catch(handleProfileError);
```

В таком коде нормальный результат из getJSON() по-прежнему передается displayUserProfile(), но любая ошибка в getJSON() или в displayUserProfile() (в том числе любые исключения, сгенерированные displayUserProfile()) передается handleProfileError(). Метод catch() является просто сокращением для вызова then() с первым аргументом, равным null, и указанной функцией обработчика ошибок в качестве второго аргумента.

О методе catch() и такой идиоме обработки ошибок еще пойдет речь во время обсуждения цепочек объектов Promise в следующем подразделе.

Терминология, связанная с объектами Promise

Прежде чем мы продолжим обсуждение объектов Promise, имеет смысл сделать паузу с целью определения ряда терминов. Когда мы не программируем и говорим о человеческих обещаниях, то констатируем, что обещание "сдержано" или "нарушено". При обсуждении объектов Promise в JavaScript эквивалентными терминами будут "удовлетворено" и "отклонено". Представьте, что вы вызвали метод then() объекта Promise и передали ему две функции обратного вызова. Мы говорим, что объект Promise (обещание) был *удовлетворен* (fulfilled), если и когда вызван первый обратный вызов. И мы говорим, что объект Promise (обещание) был *отклонен* (rejected), если и когда вызван второй обратный вызов. Если объект Promise ни удовлетворен, ни отклонен, тогда он считается *ожидающим решения* (pending). А после того, как объект Promise удовлетворен или отклонен, мы говорим, что он является *урегулированным* (settled). Имейте в виду, что объект Promise не может быть одновременно удовлетворенным *и* отклоненным. Как только объект Promise становится урегулированным, он никогда не изменится с удовлетворенного на отклоненный и наоборот.

Вспомните, что в начале раздела мы определяли Promise как объект, который представляет *результат* асинхронной операции. Важно не забывать о том, что объекты Promise являются не просто абстрактными способами регистрации обратных вызовов для запуска, когда заканчивает работу какой-то асинхронный код — они представляют результаты этого асинхронного кода. Если асинхронный код завершился нормально (и объект Promise был удовлетворен), тогда результатом по существу будет возвращаемое значение кода. Если же асинхронный код не завершился нормально (и объект

Promise был отклонен), то результатом окажется объект Error или какое-то другое значение, которое код мог бы сгенерировать, если бы он не был асинхронным. Любой объект Promise, который был урегулирован, имеет ассоциированное с ним значение, и данное значение не будет изменяться. Если объект Promise удовлетворен, тогда значением является возвращаемое значение, которое передается функции обратного вызова, зарегистрированной как первый аргумент then(). Если объект Promise отклонен, то значением является ошибка определенного рода, которая передается функции обратного вызова, зарегистрированной с помощью catch() или второго аргумента then().

Причина, по которой я хочу уточнить терминологию, касающуюся Promise, связана с тем, что объекты Promise также могут быть *разрешенными* (resolved). Разрешенное состояние легко спутать с удовлетворенным или урегулированным состоянием, но они не в точности одинаковы. Понимание разрешенного состояния — ключевой аспект глубокого понимания объектов Promise, и я вернусь к нему во время обсуждения цепочек Promise далее в главе.

13.2.2. Выстраивание объектов Promise в цепочки

Одним из наиболее важных преимуществ объектов Promise является тот факт, что они предлагают естественный способ выражения последовательности асинхронных операций в виде линейной цепочки вызовов метода then() без необходимости во вкладывании каждой операции внутрь обратного вызова предыдущей операции. Например, ниже показана гипотетическая цепочка Promise:

```
fetch(documentURL)                    // Сделать HTTP-запрос.
  .then(response => response.json())// Затребовать тело JSON ответа.
  .then(document => {   // Когда получены разобранные данные JSON,
    return render(document);    //отобразить документ пользователю.
  })
  .then(rendered => {   // Когда получен визуализированный документ,
    cacheInDatabase(rendered); // кешировать его в локальной
                               // базе данных.
  })
  .catch(error => handle(error)); // Обработать любые возникшие ошибки
```

Приведенный код демонстрирует, каким образом цепочка объектов Promise способна выразить последовательность асинхронных операций. Однако мы не собираемся обсуждать эту конкретную цепочку Promise, но продолжим исследовать идею применения цепочек Promise для выполнения HTTP-запросов.

Ранее в главе мы использовали объект XMLHttpRequest для выполнения HTTP-запроса в коде JavaScript. Данный странно именованный объект имеет старый неуклюжий API-интерфейс и почти полностью был заменен более новым API-интерфейсом fetch(), основанным на Promise (см. подраздел 15.11.1).

В своей простейшей форме новый API-интерфейс для работы с HTTP представляет собой всего лишь функцию `fetch()`. Ей передается URL, а возвращает она объект `Promise`, который удовлетворяется, когда начинает поступать HTTP-ответ, и становятся доступными состояние и заголовки HTTP:

```
fetch("/api/user/profile").then(response => {
    // Когда объект Promise разрешен, мы имеем состояние и заголовки.
    if (response.ok &&
        response.headers.get("Content-Type") === "application/json") {
      // Что здесь можно сделать? Фактически у нас пока нет тела ответа.
    }
});
```

Когда возвращенный функцией `fetch()` объект `Promise` удовлетворен, он передает объект ответа функции, переданной методу `then()`. Такой объект ответа предоставляет доступ к состоянию и заголовкам ответа плюс определяет методы вроде `text()` и `json()`, которые открывают доступ к телу ответа в текстовой форме и в форме JSON. Но хотя первоначальный объект `Promise` удовлетворен, тело ответа возможно еще не поступило. Следовательно, методы `text()` и `json()` для доступа к телу ответа сами возвращают объекты `Promise`. Вот как выглядит наивный способ применения функции `fetch()` и метода `response.json()` для получения тела HTTP-ответа:

```
fetch("/api/user/profile").then(response => {
    response.json().then(profile => {  // Затребовать тело в форме JSON.
        // Когда поступит тело ответа, оно будет автоматически
        // разобрано как JSON и передано этой функции.
        displayUserProfile(profile);
    });
});
```

Такой способ использования объектов `Promise` является наивным, поскольку мы вкладываем эти объекты подобно обратным вызовам, что противоречит их цели. Предпочтительная идиома предусматривает применение объектов `Promise` в последовательной цепочке с помощью кода следующего вида:

```
fetch("/api/user/profile")
    .then(response => {
        return response.json();
    })
    .then(profile => {
        displayUserProfile(profile);
    });
```

Давайте взглянем на вызовы методов в коде, проигнорировав передаваемые им аргументы:

```
fetch().then().then()
```

Когда в одиночном выражении подобного рода вызывается более одного метода, мы называем его *цепочкой методов*. Нам известно, что функция `fetch()` возвращает объект `Promise`, и мы видим, что первая конструкция `.then()` в цепочке вызывает метод возвращенного объекта `Promise`. Но в цепочке есть

и вторая конструкция .then(), а это значит, что первый вызов метода then() сам обязан возвращать объект Promise.

Иногда если какой-то API-интерфейс спроектирован для использования такого выстраивания методов в цепочки, то существует только один объект, и каждый метод этого объекта возвращает сам объект, чтобы упростить построение цепочки. Тем не менее, объекты Promise работают не так. Когда мы пишем цепочку вызовов .then(), мы не регистрируем множество обратных вызовов на единственном объекте Promise. Взамен каждый вызов метода then() возвращает новый объект Promise. Этот новый объект Promise не будет удовлетворен до тех пор, пока не завершится функция, переданная методу then().

Вернемся к упрощенной форме исходной цепочки fetch(), показанной выше. Если мы где-то определим функции, передаваемые then(), то можем переписать код следующим образом:

```
fetch(theURL)           // задача 1; возвращает Promise1
   .then(callback1)     // задача 2; возвращает Promise2
   .then(callback2);    // задача 3; возвращает Promise3
```

Давайте пройдемся по приведенному коду.

1. В первой строке функция fetch() вызывается с URL. Она инициирует HTTP-запрос GET для указанного URL и возвращает объект Promise. Мы будем называть этот HTTP-запрос задачей 1, а возвращаемый объект Promise — Promise1.
2. Во второй строке мы вызываем метод then() объекта Promise1, передавая ему функцию callback1, которая должна вызываться, когда удовлетворен объект Promise1. Метод then() где-то сохраняет наш обратный вызов и затем возвращает новый объект Promise. Мы назовем новый объект Promise, возвращенный на данном шаге, Promise2, и будем говорить, что "задача 2" начинается при вызове callback1.
3. В третьей строке мы вызываем метод then() объекта Promise2, передавая ему функцию callback2, которая должна вызываться, когда удовлетворен объект Promise2. Метод then() запоминает наш обратный вызов и возвращает еще один объект Promise. Мы будем говорить, что "задача 3" начинается, когда вызывается callback2. Мы можем назначить последнему объекту Promise имя Promise3, но в действительности он в имени не нуждается, т.к. вообще не будет использоваться.
4. Все три предшествующих шага происходят синхронно при первом выполнении выражения. Теперь у нас есть асинхронная пауза, пока HTTP-запрос, инициированный на шаге 1, передается через Интернет.
5. Со временем начинает поступать HTTP-ответ. Асинхронная часть вызова fetch() помещает состояние и заголовки HTTP внутрь объекта ответа и удовлетворяет объект Promise1 с этим объектом ответа в качестве значения.
6. Когда объект Promise1 удовлетворен, его значение (объект ответа) передается функции callback1() и начинается задача 2. Работа этой задачи,

имеющей на входе объект Response, заключается в том, чтобы получить тело ответа ка объект JSON.

7. Предположим, что задача 2 завершается нормально и есть возможность провести разбор тела HTTP-ответа для получения объекта JSON. Этот объект JSON используется для удовлетворения объекта Promise2.

8. Значение, которое удовлетворяет объект Promise2, становится входными данными задачи 3, когда оно передается функции callback2(). Теперь задача 3 отображает данные пользователю каким-то неоговоренным способом. Когда задача 3 завершается (предположительно нормально), удовлетворяется объект Promise3. Но поскольку мы ничего не делаем с объектом Promise3, то ничего и не происходит, когда этот объект Promise урегулируется, а цепочка асинхронных вычислений заканчивается в данной точке.

13.2.3. Разрешение объектов Promise

Во время объяснения цепочки Promise для извлечения URL мы говорили об объектах Promise1, Promise2 и Promise3. Но на самом деле задействован также четвертый объект Promise, и это подводит нас к важному обсуждению того, что означается для объекта Promise быть "разрешенным".

Вспомните, что функция fetch() возвращает объект Promise, который в случае удовлетворения передает объект ответа в зарегистрированную нами функцию обратного вызова. Объект ответа имеет методы .text(), .json() и другие методы для получения тела HTTP-ответа в разнообразных формах. Но из-за того, что тело может еще не поступить, такие методы обязаны возвращать объекты Promise. В приведенном выше примере задача 2 вызывает метод .json() и возвращает его значение. Это четвертый объект Promise и он является возвращаемым значением функции callback1().

Давайте перепишем код извлечения URL еще раз в многословном и неидиоматичном виде, который позволяет сделать явными обратные вызовы и объекты Promise:

```
function c1(response) {                 // Обратный вызов 1
    let p4 = response.json();
    return p4;                          // Возвращает Promise4
}

function c2(profile) {                  // Обратный вызов 2
    displayUserProfile(profile);
}

let p1 = fetch("/api/user/profile");// Promise1, задача 1
let p2 = p1.then(c1);               // Promise2, задача 2
let p3 = p2.then(c2);               // Promise3, задача 3
```

Чтобы цепочки объектов Promise работали успешно, выходные данные задачи 2 должны стать входными данными задачи 3. И в рассматриваемом здесь примере входными данными задачи 3 является тело извлеченного URL, разо-

бранное как объект JSON. Но мы только что выяснили, что возвращаемое значение обратного вызова c1 представляет собой не объект JSON, а объект Promise по имени p4 для объекта JSON. Это выглядит как несоответствие, но в действительности нет: когда p1 удовлетворен, c1 вызывается и задача 2 начинается. И когда p2 удовлетворен, c2 вызывается и задача 3 начинается. Но тот факт, что задача 2 начинается, когда вызван c1, вовсе не означает, что задача 2 должна завершиться, когда произойдет возврат из c1. В конце концов, объекты Promise связаны с управлением асинхронными задачами, и если задача 2 асинхронная (а в нашем случае это так), тогда она не будет завершена ко времени возврата из обратного вызова.

Теперь все готово к обсуждению последних деталей, которые вам необходимо понять, чтобы по-настоящему освоить объекты Promise. Когда вы передаете обратный вызов c методу then(), то then() возвращает объект Promise по имени p и организует асинхронный вызов c в более позднее время. Обратный вызов выполняет какие-то вычисления и возвращает значение v. Когда происходит возврат из обратного вызова, p *разрешается* со значением v.

Когда объект Promise разрешен со значением, которое само не является Promise, он немедленно удовлетворяется этим значением. Таким образом, если c возвращает не объект Promise, тогда это возвращаемое значение становится значением p, p удовлетворяется и все готово. Но если возвращаемое значение v представляет собой объект Promise, то p будет *разрешенным, но пока не удовлетворенным*.

На данном этапе p не может быть урегулирован, пока не урегулируется объект Promise по имени v. Если v удовлетворен, тогда p будет удовлетворен тем же самым значением. Если v отклонен, тогда p будет отклонен по той же самой причине. Вот что означает "разрешенное" состояние объекта Promise: объект Promise стал ассоциированным с другим объектом Promise, или "заблокированным" им. Мы пока не знаем, будет p удовлетворен или отклонен, но наш обратный вызов c больше это не контролирует. Объект p является "разрешенным" в том смысле, что его судьба теперь полностью зависит от того, что произойдет с объектом Promise по имени v.

Вернемся к нашему примеру с извлечением URL. Когда c1 возвращает p4, то p2 разрешается. Но быть разрешенным — вовсе не то же самое, что быть удовлетворенным, и потому задача 3 пока не начинается. Когда полное тело HTTP-запроса становится доступным, то метод .json() может провести его разбор и применить разобранное значение для удовлетворения p4. Когда p4 удовлетворен, также автоматически удовлетворяется и p2, с тем же разобранным значением JSON. В этой точке разобранный объект JSON передается c2 и задача 3 начинается.

В настоящем разделе была изложена, вероятно, одна из самых сложных для понимания частей JavaScript, и возможно вам понадобится прочитать его несколько раз. На рис. 13.1 процесс представлен в визуальной форме, которая может дополнительно прояснить его.

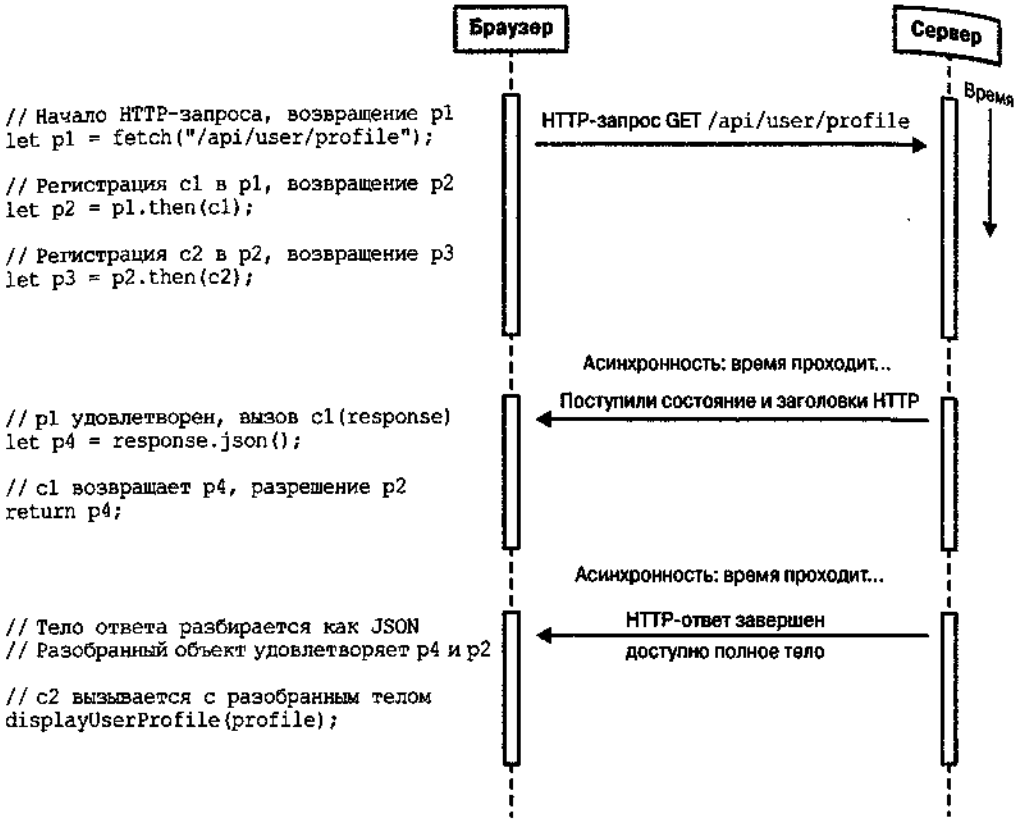


Рис. 13.1. Извлечение URL с помощью объектов `Promise`

13.2.4. Дополнительные сведения об объектах `Promise` и ошибках

Ранее в главе вы видели, что методу `.then()` можно передавать вторую функцию обратного вызова, которая будет вызываться в случае, если объект `Promise` отклонен. Когда подобное произойдет, аргументом второй функции обратного вызова будет значение (обычно объект `Error`), представляющее причину отклонения. Также выяснилось, что два обратных вызова передают методу `.then()` редко (и это даже неидиоматично). Взамен ошибки, связанные с `Promise`, как правило, обрабатывают добавлением к цепочке `Promise` вызова метода `.catch()`. Теперь, после исследования цепочек `Promise`, мы можем возвратиться к обработке ошибок и обсудить ее более полно. Перед обсуждением я хотел бы подчеркнуть, что в асинхронном программировании аккуратная обработка ошибок чрезвычайно важна. Если вы опустите код обработки ошибок в синхронном коде, то, по крайней мере, получите исключение и трассировку стека, которую можно использовать для выяснения, что пошло не так. В случае применения асинхронного кода о необработанных исключениях часто не сообщается и ошибки могут происходить молча, что значительно затрудняет процесс отладки. Хорошая новость в том, что метод `.catch()` облегчает обработку ошибок при работе с объектами `Promise`.

Методы .catch() и .finally()

Метод .catch() объекта Promise — это просто сокращенный способ вызова .then() с null в первом аргументе и обратным вызовом для обработки ошибок во втором аргументе. Для любого объекта Promise по имени p и обратного вызова по имени c следующие две строки эквивалентны:

```
p.then(null, c);
p.catch(c);
```

Сокращение .catch() предпочтительнее, потому что проще и его имя совпадает с конструкцией catch в операторе обработки исключений try/catch. Как обсуждалось ранее, нормальные исключения не работают с асинхронным кодом. Метод .catch() объектов Promise является альтернативой, которая работает с асинхронным кодом. Когда что-то пошло не так в синхронном коде, то мы можем сказать, что исключение "поднимается словно пузырек вверх по стеку вызовов", пока не найдет блок catch. В асинхронной цепочке объектов Promise сопоставимой метафорой может быть то, что ошибка "стекает вниз по цепочке", пока не найдет вызов .catch().

В ES2018 объекты Promise также определяют метод .finally(), цель которого аналогична цели конструкции finally в операторе try/catch/finally. Если вы добавите к своей цепочке Promise вызов .finally(), тогда переданный методу .finally() обратный вызов будет вызван, когда Promise урегулируется. Ваш обратный вызов будет вызван в случае удовлетворения или отклонения Promise, но поскольку аргументы не передаются, то вы не сможете узнать, был Promise удовлетворен или отклонен. Но если вам в любом случае нужно выполнить код очистки (скажем, закрывающий открытые файлы или сетевые подключения), тогда обратный вызов .finally() — идеальный способ сделать это. Подобно .then() и .catch(), метод .finally() возвращает новый объект Promise. Возвращаемое значение обратного вызова .finally() обычно игнорируется, а объект Promise, возвращенный .finally(), как правило, будет разрешен или отклонен с тем же значением, что и объект Promise, с которым вызывался метод .finally() при разрешении или отклонении. Однако если в обратном вызове .finally() генерируется исключение, то возвращенный методом .finally() объект Promise будет отклонен с таким значением.

В коде извлечения URL, который мы изучали в предшествующих разделах, никакой обработки ошибок не было предусмотрено. Давайте исправим это, написав более реалистичную версию кода:

```
fetch("/api/user/profile") // Начать HTTP-запрос.
  .then(response => { //Вызывается, когда готовы состояние и заголовки.
    if (!response.ok) {  // Если мы получаем ошибку
                         // 404 Not Found или похожую.
      return null;       //Возможно, пользователь вышел из системы;
                         // возвратить профиль null.
    }
    // Проверить заголовки, чтобы удостовериться,
    // что сервер отправил нам JSON.
    // Если нет, тогда наш сервер неисправен, а это серьезная ошибка!
```

```
    let type = response.headers.get("content-type");
    if (type !== "application/json") {
      throw new TypeError(`Expected JSON, got ${type}`);
                    // Ожидался JSON, но получен другой тип
    }
    // Если мы попали сюда, то получили состояние 2xx
    // и тип содержимого JSON, поэтому можем уверенно возвратить
    // объект Promise для тела ответа как объект JSON.
    return response.json();
  })
  .then(profile => { //Вызывается с разобранным телом ответа или null
    if (profile) {
      displayUserProfile(profile);
    }
    else {          // Если мы получили ошибку 404 выше
                    // и возвратили null, то окажемся здесь.
      displayLoggedOutProfilePage();
    }
  })
  .catch(e => {
    if (e instanceof NetworkError) {
      // fetch() может потерпеть такую неудачу,
      // если исчезло подключение к Интернету.
      displayErrorMessage("Check your internet connection.");
                    // Проверьте свое подключение к Интернету.
    }
    else if (e instanceof TypeError) {
      // Это происходит в случае генерации TypeError выше.
      displayErrorMessage("Something is wrong with our server!");
                    // Что-то не так с нашим сервером!
    }
    else {
      // Это должна быть непредвиденная ошибка какого-то вида.
      console.error(e);
    }
  });
```

Давайте проанализируем код, посмотрев на то, что происходит, когда что-то пошло не так. Мы будем использовать схему именования, применяемую ранее: `p1` — объект `Promise`, возвращенный вызовом `fetch()`. Затем `p2` — объект `Promise`, возвращенный первым вызовом `.then()`, а `c1` — обратный вызов, который мы передаем этому вызову `.then()`. Далее `p3` — объект `Promise`, возвращенный вторым вызовом `.then()`, а `c2` — обратный вызов, передаваемый этому вызову. Наконец, `c3` — обратный вызов, который мы передаем вызову `.catch()`. (Такой вызов возвращает объект `Promise`, но ссылаться на него по имени нет никакой необходимости.)

Первое, что может отказать — сам запрос `fetch()`. Если сетевое подключение не работает (или HTTP-запрос не может быть выполнен по какой-то другой причине), тогда объект `Promise` по имени `p1` будет отклонен с объектом `NetworkError`. Мы не передавали вызову `.then()` функцию обратного вызо-

ва для обработки ошибок во втором аргументе, поэтому p2 также отклоняется с тем же самым объектом NetworkError. (Если бы мы передали обработчик ошибок первому вызову .then(), то обработчик ошибок вызвался бы и в случае нормального возвращения p2 был бы разрешен и/или удовлетворен с возвращаемым значением данного обработчика.) Тем не менее, без обработчика p2 отклоняется и затем по той же причине отклоняется p3. В этой точке вызывается обратный вызов для обработки ошибок c3 и внутри него выполняется код, специфичный для NetworkError.

Наш код может потерпеть неудачу и по-другому: когда HTTP-запрос возвращает 404 Not Found (не найдено) либо иную ошибку HTTP. Они являются допустимыми HTTP-ответами, так что вызов fetch() не считает их ошибками. Функция fetch() инкапсулирует 404 Not Found в объекте ответа и удовлетворяет p1 с таким объектом, приводя к вызову c1. Наш код в c1 проверяет свойство ok объекта ответа для определения, что он не получил нормальный HTTP-ответ, и обрабатывает данный случай, просто возвращая null. Поскольку это возвращаемое значение не является объектом Promise, оно сразу удовлетворяет p2 и с ним вызывается c2. Наш код в c2 явно проверяет и обрабатывает ложные значения, отображая пользователю другой результат. Это случай, когда мы трактуем ненормальное условие как отличающееся от ошибки и обрабатываем его без действительного использования обработчика ошибок.

Более серьезная ошибка возникает в c1, если мы получаем нормальный код HTTP-ответа, но заголовок Content-Type (тип содержимого) не установлен надлежащим образом. Наш код ожидает ответ в формате JSON, так что если сервер посылает взамен HTML, XML или простой текст, то у нас возникнет проблема. В c1 есть код для проверки заголовка Content-Type. Если заголовок некорректен, то это трактуется как невосстановимая проблема и генерируется TypeError. Когда обратный вызов, переданный .then() (или .catch()), генерирует какое-то значение, объект Promise, который был возвращаемым значением вызова .then(), отклоняется со сгенерированным значением. В данном случае код в c1, генерирующий TypeError, приводит к отклонению p2 с этим объектом TypeError. Поскольку мы не указали обработчик ошибок для p2, то p3 также будет отклонен. c2 не вызывается и объект TypeError передается c3, где имеется код для явной проверки и обработки ошибки такого типа.

В приведенном коде полезно отметить несколько моментов. Первым делом обратите внимание, что объект ошибки, сгенерированный с помощью обыкновенного синхронного оператора throw, в итоге обрабатывается асинхронно посредством вызова метода .catch() в цепочке Promise. Это должно прояснить, почему сокращенный метод предпочтительнее передачи второго аргумента методу .then(), а также по какой причине настолько идиоматично заканчивать цепочки Promise вызовом .catch().

Прежде чем оставить тему обработки ошибок, я хочу отметить, что хотя идиоматично завершать каждую цепочку Promise вызовом .catch() для очистки (или, по крайней мере, регистрации) любых ошибок, которые в ней происходили, также совершенно допустимо применять .catch() где-то в другом месте цепочки Promise. Если на одной из стадий в вашей цепочке Promise мо-

жет возникнуть отказ с ошибкой, которая является восстановимой и не должна препятствовать выполнению остатка цепочки, тогда можете вставить вызов `.catch()` в цепочку, получив примерно такой код:

```
startAsyncOperation()
    .then(doStageTwo)
    .catch(recoverFromStageTwoError)
    .then(doStageThree)
    .then(doStageFour)
    .catch(logStageThreeAndFourErrors);
```

Вспомните, что обратный вызов, передаваемый `.catch()`, будет вызываться, только если обратный вызов на предыдущей стадии сгенерировал ошибку. Если этот обратный вызов выполнился нормально, то обратный вызов `.catch()` пропускается и возвращаемое значение предыдущего обратного вызова становится входными данными следующего обратного вызова `.then()`. Также вспомните, что обратные вызовы `.catch()` предназначены не только для сообщения об ошибках, но для обработки и восстановления после возникновения ошибок. После того, как ошибка была передана обратному вызову `.catch()`, она прекращает распространяться вниз по цепочке `Promise`. Обратный вызов `.catch()` может сгенерировать новую ошибку, но если происходит нормальный возврат из него, то возвращаемое значение используется для разрешения и/или удовлетворения ассоциированного объекта `Promise`, и ошибка перестает распространяться.

Давайте более конкретно: если в предыдущем примере кода либо `startAsyncOperation()`, либо `doStageTwo()` генерирует ошибку, тогда будет вызвана функция `recoverFromStageTwoError()`. В случае нормального возврата из функции `recoverFromStageTwoError()` ее возвращаемое значение будет передано `doStageThree()` и асинхронная операция благополучно продолжится. С другой стороны, если функции `recoverFromStageTwoError()` не удалось выполнить восстановление, то она сама сгенерирует ошибку (или повторно сгенерирует переданную ей ошибку). В такой ситуации ни `doStageThree()`, ни `doStageFour()` не вызывается, а ошибка, сгенерированная `recoverFromStageTwoError()`, будет передана `logStageThreeAndFourErrors()`.

Иногда в сложных сетевых средах ошибки могут возникать более или менее случайным образом, и вполне целесообразно обрабатывать их, просто повторяя асинхронный запрос. Пусть написана операция на основе `Promise` для запрашивания базы данных:

```
queryDatabase()
    .then(displayTable)
    .catch(displayDatabaseError);
```

Теперь предположим, что из-за кратковременных проблем с загрузкой сети операция терпит неудачу приблизительно в 1% случаев. Простое решение может предусматривать повторение запроса с помощью вызова `.catch()`:

```
queryDatabase()
    .catch(e => wait(500).then(queryDatabase)) //При отказе подождать
                                               // и повторить.
    .then(displayTable)
    .catch(displayDatabaseError);
```

Если гипотетические отказы по-настоящему случайны, тогда добавление этой одной строки кода должно сократить частоту появления ошибок с 1% до 0,01%.

Возврат из обратного вызова объекта Promise

Давайте в последний раз вернемся к рассмотренному ранее примеру извлечения URL и обсудим обратный вызов c1, который мы передавали первому вызову .then(). Обратите внимание, что обратный вызов c1 может завершиться тремя путями. Может произойти нормальный возврат с объектом Promise, возвращенным вызовом .json(). Это приводит к разрешению p2, но будет ли данный объект Promise удовлетворен или отклонен, зависит от того, что случится с вновь возвращенным объектом Promise. Кроме того, возврат из c1 может также нормально произойти со значением null, что вызывает немедленное удовлетворение p2. Наконец, c1 может завершиться с генерацией ошибки, приводя к отклонению p2. Существуют три возможных исхода для Promise, и код в c1 демонстрирует, каким образом обратный вызов может инициировать каждый исход.

В цепочке Promise значение, возвращенное (или сгенерированное) на одной стадии цепочки, становится входными данными следующей стадии цепочки, поэтому важно делать это правильно. На практике распространенным источником ошибок, связанных с цепочками Promise, является отсутствие возврата значения из функции обратного вызова, что дополнительно усугубляется наличием сокращенного синтаксиса стрелочных функций JavaScript. Возьмем следующую строку кода, которая встречалась ранее:

```
.catch(e => wait(500).then(queryDatabase))
```

Вспомните из главы 8, что стрелочные функции допускают много сокращений. Поскольку имеется в точности один аргумент (значение ошибки), мы можем опустить круглые скобки. Так как тело функции представляет собой одиночное выражение, мы можем опустить фигурные скобки вокруг тела, и значение выражения станет возвращаемым значением функции. По причине допустимости таких сокращений предыдущий код корректен. Но взгляните на показанное ниже с виду безобидное изменение:

```
.catch(e => { wait(500).then(queryDatabase) })
```

Добавив фигурные скобки, мы утрачиваем автоматический возврат. Теперь функция возвращает undefined, а не объект Promise, и тогда следующая стадия в цепочке Promise будет вызвана с undefined в качестве входных данных вместо результата повторного запроса. Такую тонкую ошибку зачастую нелегко выявить.

13.2.5. Параллельное выполнение нескольких асинхронных операций с помощью Promise

Мы потратили много времени на обсуждение цепочек Promise для последовательного выполнения асинхронных шагов более крупной асинхронной операции. Однако иногда желательно параллельно выполнять сразу несколько асинхронных операций. Это позволяет делать функция Promise.all(), которая принимает на входе массив объектов Promise и возвращает объект Promise. Возвращенный объект Promise будет отклонен, если отклонен любой из входных объектов Promise. В противном случае он будет удовлетворен с массивом значений, которыми удовлетворены все входные объекты Promise. Итак, например, если вы хотите извлечь текстовое содержимое множества URL, то можете применить код следующего вида:

```
// Мы начинаем с массива URL.
const urls = [ /* ноль или большее количество URL */ ];

// И преобразуем его в массив объектов Promise.
promises = urls.map(url => fetch(url).then(r => r.text()));

//Теперь получаем объект Promise для запуска всех объектов Promise параллельно
Promise.all(promises)
    .then(bodies => { /* делать что-нибудь с массивом строк */ })
    .catch(e => console.error(e));
```

Функция Promise.all() обладает чуть большей гибкостью, чем было описано ранее. Входной массив может содержать и объекты Promise, и значения, отличающиеся от Promise. Когда элемент массива не является объектом Promise, он трактуется, как если бы он был значением уже удовлетворенного объекта Promise, и просто копируется без изменений в выходной массив.

Объект Promise, возвращенный Promise.all(), отклоняется при отклонении любого из входных объектов Promise. Это происходит сразу же после первого отклонения и может случиться так, что остальные входные объекты все еще будут ожидать решения. Функция Promise.allSettled() в ES2020 принимает массив входных объектов Promise и подобно Promise.all() возвращает объект Promise. Но Promise.allSettled() никогда не отклоняет возвращенный объект Promise и не удовлетворяет его до тех пор, пока не будут урегулированы все входные объекты Promise. Объект Promise разрешается в массив объектов, по одному на каждый входной объект Promise. Каждый возвращенный объект имеет свойство status (состояние), установленное в "fulfilled" (удовлетворен) или "rejected" (отклонен). Если состоянием является "fulfilled", тогда объект будет также иметь свойство value (значение), которое дает значение удовлетворения. А если состоянием оказывается "rejected", то объект будет также иметь свойство reason (причина), которое дает ошибку или значение отклонения соответствующего объекта Promise:

```
Promise.allSettled([Promise.resolve(1), Promise.reject(2), 3]).then(results =>{
    results[0]      // => { status: "fulfilled", value: 1 }
    results[1]      // => { status: "rejected", reason: 2 }
    results[2]      // => { status: "fulfilled", value: 3 }
});
```

Изредка у вас может возникать необходимость запустить сразу несколько объектов Promise, но заботиться только о значении первого удовлетворенного объекта. В таком случае вы можете использовать Promise.race() вместо Promise.all(). Функция Promise.race() возвращает объект Promise, который удовлетворен или отклонен, когда был удовлетворен или отклонен первый из объектов Promise во входном массиве. (Либо если во входном массиве присутствуют значения не Promise, то он просто возвращает первое из них.)

13.2.6. Создание объектов Promise

Мы применяли функцию fetch(), возвращающую Promise, во многих предшествующих примерах, потому что она является одной из простейших встроенных в веб-браузеры функций, которые возвращают объект Promise. В нашем обсуждении объектов Promise мы также полагались на гипотетические функции getJSON() и wait(), возвращающие Promise. Функции, написанные для возвращения объектов Promise, действительно очень полезны, и в настоящем подразделе будет показано, как создавать собственные API-интерфейсы, основанные на Promise. В частности, мы рассмотрим реализации getJSON() и wait().

Объекты Promise, основанные на других объектах Promise

Написать функцию, возвращающую объект Promise, довольно легко при наличии другой функции такого рода, с которой можно начать. Имея объект Promise, вы всегда можете создать (и возвратить) новый объект Promise, вызвав .then(). Таким образом, если мы воспользуемся существующей функцией fetch() в качестве отправной точки, тогда можем написать getJSON() примерно так:

```
function getJSON(url) {
   return fetch(url).then(response => response.json());
}
```

Код тривиален, потому что объект ответа API-интерфейса fetch() имеет заранее определенный метод json(). Метод json() возвращает объект Promise, который мы возвращаем из нашего обратного вызова (обратный вызов представлен как стрелочная функция с телом, содержащим единственное выражение, поэтому возврат происходит неявно), так что объект Promise, возвращенный getJSON(), разрешается в объект Promise, возвращенный response.json(). Когда этот объект Promise удовлетворяется, с тем же самым значением удовлетворяется и объект Promise, возвращенный getJSON(). Обратите внимание, что в приведенной реализации getJSON() обработка ошибок не предусмотрена. Вместо проверки response.ok и заголовка Content-Type мы просто позволяем методу json() отклонить возвращенный объект Promise с SyntaxError, если тело ответа не может быть разобрано как JSON.

Давайте напишем еще одну функцию, возвращающую Promise, но на этот раз применим getJSON() в качестве источника начального объекта Promise:

```
function getHighScore() {
  return getJSON("/api/user/profile").then(profile=>profile.highScore);
}
```

Мы предполагаем, что функция `getHighScore()` входит в состав какой-то веб-игры и URL вида `/api/user/profile` возвращает структуру данных в формате JSON, которая включает свойство `highScore`.

Объекты `Promise`, основанные на синхронных значениях

Иногда вам может понадобиться реализовать существующий API-интерфейс на основе `Promise` и возвращать объект `Promise` из функции, даже если выполняемое вычисление в действительности не требует никаких асинхронных операций. В таком случае статические методы `Promise.resolve()` и `Promise.reject()` будут делать то, что вас интересует. Метод `Promise.resolve()` принимает значение в своем единственном аргументе и возвращает объект `Promise`, который будет немедленно (но асинхронно) удовлетворен в это значение. Аналогично метод `Promise.reject()` принимает значение в единственном аргументе и возвращает объект `Promise`, который будет отклонен с этим значением в качестве причины. (Ради ясности: объекты `Promise`, возвращенные статическими методами `Promise.resolve()` и `Promise.reject()`, не являются удовлетворенными или отклоненными сразу по возврату, но будут удовлетворены или отклонены немедленно после того, как завершит выполнение текущая синхронная порция кода. Обычно подобное происходит в пределах нескольких миллисекунд, если только не скопилось много незаконченных асинхронных задач, ожидающих выполнения.)

Вспомните из подраздела 13.2.3, что разрешенный объект `Promise` — не то же самое, что и удовлетворенный объект `Promise`. Когда мы вызываем `Promise.resolve()`, то обычно передаем удовлетворенное значение, чтобы создать объект `Promise`, который очень скоро будет удовлетворен с этим значением. Тем не менее, метод не называется `Promise.fulfill()`. Если вы передадите объект `Promise` по имени `p1` методу `Promise.resolve()`, то он возвратит новый объект `Promise` по имени `p2`, который немедленно разрешается, но не будет удовлетворен или отклонен до тех пор, пока `p1` не станет удовлетворенным или отклоненным.

Возможно, но необычно, написать функцию, основанную на `Promise`, где значение вычисляется синхронно и возвращается асинхронно с помощью `Promise.resolve()`. Однако наличие синхронных особых случаев внутри асинхронной функции — довольно распространенная практика, и вы можете обрабатывать такие особые случаи посредством `Promise.resolve()` и `Promise.reject()`. Скажем, если вы обнаружили условие ошибки (такое как неправильные значения аргументов) до начала асинхронной операции, то можете сообщить об ошибке за счет возвращения объекта `Promise`, созданного с помощью `Promise.reject()`. (В данном случае вы также могли бы просто синхронно сгенерировать ошибку, но это считается плохой формой, поскольку вызывающему коду для обработки ошибок пришлось бы записывать синхронную конструкцию `catch` и использовать асинхронный метод `.catch()`.)

Наконец, метод Promise.resolve() временами удобен для создания начального объекта Promise в цепочке объектов Promise. Далее еще будут показаны примеры его применения подобным образом.

Объекты Promise с нуля

Для методов getJSON() и getHighScore() мы первым делом вызывали существующую функцию, чтобы получить начальный объект Promise, а также создавали и возвращали новый объект Promise, вызывая метод .then() начального объекта Promise. Но как насчет написания функции, возвращающей Promise, когда вы не можете использовать другую функцию, возвращающую Promise, в качестве отправной точки? В этом случае вы применяете конструктор Promise() для создания нового объекта Promise, который находится под полным вашим контролем. Вот как все работает: вы вызываете конструктор Promise() и передаете ему функцию в его единственном аргументе. Передаваемая функция должна быть написана так, чтобы ожидать два параметра, которые по соглашению обязаны иметь имена resolve и reject. Конструктор синхронно вызывает вашу функцию с аргументами для параметров resolve и reject. После вызова вашей функции конструктор Promise() возвращает вновь созданный объект Promise. Возвращенный объект Promise находится под контролем функции, переданной вами конструктору. Эта функция должна выполнить определенную асинхронную операцию и затем вызвать функцию resolve, чтобы разрешить или удовлетворить возвращенный объект Promise, или вызвать функцию reject, чтобы отклонить возвращенный объект Promise. Ваша функция не обязана быть асинхронной: она может вызывать resolve или reject синхронным образом, но объект Promise по-прежнему будет асинхронно разрешен, удовлетворен или отклонен.

Понять, что собой представляют функции, передаваемые в функцию, которая передается конструктору, может быть нелегко, просто прочитав о них. Надо надеяться, что с помощью ряда примеров ситуация прояснится. Ниже показано, как написать основанную на Promise функцию wait(), которую мы использовали в различных примерах ранее в главе:

```
function wait(duration) {
    // Создать и возвратить новый объект Promise.
    return new Promise((resolve, reject) => { // Это контролирует
                                              // объект Promise.
  //Если значение аргумента недопустимо, тогда отклонить объект Promise
      if (duration < 0) {
        reject(new Error("Time travel not yet implemented"));
                        // Путешествия во времени пока не осуществимы
      }
      // В противном случае асинхронно ожидать и затем разрешить
      // объект Promise.
      //setTimeout будет вызывать resolve() без аргументов, а это значит,
      // что объект Promise будет удовлетворен в значение undefined.
      setTimeout(resolve, duration);
    });
}
```

Обратите внимание, что пара функций, которые вы применяете для управления судьбой объекта Promise, созданного посредством конструктора Promise(), называются resolve() и reject(), а не fulfill() и reject(). Если вы передадите объект Promise функции resolve(), то возвращенный Promise будет разрешен в новый Promise. Тем не менее, зачастую вы будете передавать значение, отличающееся от Promise, что удовлетворяет возвращенный Promise этим значением.

В примере 13.1 представлен еще один способ использования конструктора Promise(). В нем реализована наша функция getJSON() для применения в Node, где API-интерфейс fetch() не является встроенным. Вспомните, что глава начиналась с обсуждения асинхронных обратных вызовов и событий. В примере 13.1 используются обратные вызовы и обработчики событий, следовательно, он служит хорошей демонстрацией того, как можно реализовывать API-интерфейсы, основанные на Promise, поверх других стилей асинхронного программирования.

Пример 13.1. Асинхронная функция getJSON()

```
const http = require("http");

function getJSON(url) {
   // Создать и возвратить новый объект Promise.
   return new Promise((resolve, reject) => {
      // Запустить HTTP-запрос GET для указанного URL.
      request = http.get(url, response => {  // Вызывается, когда начинает
                                             // поступать ответ.
         //Отклонить объект Promise, если состояние HTTP указывает на ошибку
         if (response.statusCode !== 200) {
            reject(new Error(`HTTP status ${response.statusCode}`));
            response.resume();  // Не допустить утечки памяти.
         }
         // И отклонить, если заголовки ответа ошибочны.
         else if (response.headers["content-type"] !== "application/json") {
            reject(new Error("Invalid content-type"));
            response.resume();  // Не допустить утечки памяти.
         }
         else {
            // В противном случае зарегистрировать события
            // для чтения тела ответа.
            let body = "";
            response.setEncoding("utf-8");
            response.on("data", chunk => { body += chunk; });
            response.on("end", () => {
               // Когда тело ответа закончило поступление,
               // попытаться его разобрать.
               try {
                  let parsed = JSON.parse(body);
                  // Если тело разобрано успешно, тогда удовлетворить
                  // объект Promise.
                  resolve(parsed);
               } catch(e) {
```

```
            // Если разбор потерпел неудачу, тогда отклонить объект Promise.
                reject(e);
            }
        });
    }
});
// Также отклонить объект Promise, если запрос потерпел неудачу
// еще до получения ответа (скажем, при сбое сетевого подключения).
request.on("error", error => {
    reject(error);
});
});
}
```

13.2.7. Последовательное выполнение нескольких асинхронных операций с помощью `Promise`

Метод `Promise.all()` упрощает параллельный запуск произвольного количества объектов `Promise`, а цепочки `Promise` облегчают выражение последовательности с фиксированным числом объектов `Promise`. Однако последовательное выполнение произвольного количества объектов сложнее. Предположим, например, что у вас есть массив URL для извлечения, но во избежание перегрузки сети вы хотите извлекать их по одному за раз. Если массив имеет произвольную длину и неизвестное содержимое, тогда вам не удастся написать цепочку `Promise` заблаговременно и потому придется строить ее динамически с помощью такого кода:

```
function fetchSequentially(urls) {
    // Здесь мы будем сохранять тела URL по мере их извлечения.
    const bodies = [];

    // Функция, возвращающая Promise, которая извлекает одно тело.
    function fetchOne(url) {
        return fetch(url)
            .then(response => response.text())
            .then(body => {
                // Мы сохраняем тело в массив и преднамеренно опускаем
                // здесь возвращаемое значение (возвращая undefined).
                bodies.push(body);
            });
    }

    // Начать с объекта Promise, который будет удовлетворен
    // прямо сейчас (со значением undefined).
    let p = Promise.resolve(undefined);

    // Пройти в цикле по желательным URL с построением цепочки Promise
    // произвольной длины и извлечением одного URL на каждой стадии цепочки
    for(url of urls) {
        p = p.then(() => fetchOne(url));
    }
```

```
    // Когда удовлетворен последний объект Promise в этой цепочке, массив
    // bodies готов. Следовательно, возвратить объект Promise для массива
    //bodies. Обратите внимание, что мы не предусмотрели ни одного обработчика
    //ошибок: мы хотим позволить ошибкам распространяться в вызывающий код.
    return p.then(() => bodies);
}
```

Располагая такой функцией `fetchSequentially()`, мы могли бы извлекать URL по одному за раз с помощью кода, во многом похожего на код параллельного извлечения, который применялся для демонстрации работы `Promise.all()`:

```
fetchSequentially(urls)
    .then(bodies => { /* делать что-нибудь с массивом строк */ })
    .catch(e => console.error(e));
```

Функция `fetchSequentially()` начинается с создания объекта `Promise`, который удовлетворен немедленно после возвращения. Затем из данного объекта `Promise` строится длинная линейная цепочка `Promise` и возвращается последний объект `Promise` в цепочке. Это сродни тому, как выстроить в ряд кости домино и опрокинуть первую из них.

Существует еще один (возможно, более элегантный) подход, который мы можем принять. Вместо того чтобы создавать объекты `Promise` заблаговременно, для каждого объекта `Promise` мы используем обратный вызов, создающий и возвращающий следующий объект `Promise`. То есть взамен создания и связывания в цепочку группы объектов `Promise` мы создаем объекты `Promise`, которые разрешаются в другие объекты `Promise`. Вместо того чтобы создавать похожую на ряд костей домино цепочку объектов `Promise` мы создаем последовательность объектов `Promise`, вложенных друг в друга, что похоже на набор матрешек. При таком подходе наш код может возвращать первый (самый внешний) объект `Promise`, зная о том, что в итоге он будет удовлетворен (или отклонен!) с таким же значением, как последний (самый внутренний) объект `Promise` в последовательности. Показанная ниже функция `promiseSequence()` реализована обобщенным образом и не является специфичной для извлечения URL. Она приводится в конце обсуждения объектов `Promise` по причине своей сложности. Тем не менее, если вы внимательно читали главу, то я надеюсь, что будете в состоянии разобраться в том, как работает данная функция. В частности, обратите внимание, что функция, вложенная внутри `promiseSequence()`, кажется, рекурсивно вызывает сама себя, но из-за того, что "рекурсивный" вызов производится через метод `then()`, в действительности никакой традиционной рекурсии не происходит.

```
// Эта функция принимает массив входных значений и функцию promiseMaker.
// Для любого входного значения x в массиве функция promiseMaker(x)
// обязана возвращать объект Promise, который будет удовлетворен
// с выходным значением.
// Функция promiseSequence() возвращает объект Promise,
// который удовлетворяется вычисленными выходными значениями.
//
```

```
// Однако вместо того чтобы создавать сразу все объекты Promise и
// позволить им выполняться параллельно, promiseSequence() запускает
// только один Promise за раз и не вызывает promiseMaker() для значения
// до тех пор, пока не будет удовлетворен предыдущий объект Promise.
function promiseSequence(inputs, promiseMaker) {
    // Создать закрытую копию массива, которую можно модифицировать.
    inputs = [...inputs];

    // Это функция, которая будет использоваться в качестве обратного
    // вызова Promise. Все работает благодаря магии псевдорекурсии.
    function handleNextInput(outputs) {
        if (inputs.length === 0) {
            // Если входных значений больше не осталось,
            // тогда возвратить массив выходных значений,
            // в конце концов, удовлетворяя данный объект Promise
            // и все предшествующие разрешенные,
            // но не удовлетворенные объекты Promise.
            return outputs;
        } else {
            // Если есть входные значения для обработки, тогда мы
            // возвратим объект Promise, разрешая текущий Promise
            // с будущим значением из нового объекта Promise.
            let nextInput = inputs.shift();     // Получить следующее
                                                // входное значение,
            return promiseMaker(nextInput)      // вычислить следующее
                                                // выходное значение,
      // затем создать новый массив outputs с новым выходным значением,
                .then(output => outputs.concat(output))
                // затем организовать "рекурсию", передавая новый
                // более длинный массив outputs.
                .then(handleNextInput);
        }
    }

    // Начать с объекта Promise, который удовлетворяется с пустым массивом,
    // и использовать определенную выше функцию как его обратный вызов.
    return Promise.resolve([]).then(handleNextInput);
}
```

Функция `promiseSequence()` преднамеренно сделана обобщенной. Мы можем применять ее для извлечения нескольких URL с помощью кода следующего вида:

```
// Для имеющегося URL возвратить объект Promise,
// который удовлетворяется с текстом тела URL.
function fetchBody(url) { return fetch(url).then(r => r.text()); }

// Использовать его для последовательной выборки множества тел URL.
promiseSequence(urls, fetchBody)
    .then(bodies => { /* делать что-нибудь с массивом строк */ })
    .catch(console.error);
```

13.3. async и await

В ES2017 появились два новых ключевых слова, async и await, которые представляют смену парадигмы в асинхронном программировании на JavaScript. Они значительно упрощают применение объектов Promise и позволяют нам писать основанный на Promise асинхронный код, выглядящий как синхронный код, который блокируется при ожидании ответов сети или других асинхронных событий. Хотя по-прежнему важно понимать, как работают объекты Promise, большая часть их сложности (а иногда даже само их присутствие!) исчезает, когда они используются с ключевыми словами async и await.

Как обсуждалось ранее в главе, что асинхронный код не может возвращать значение или генерировать исключение способом, принятым в обыкновенном синхронном коде. Вот почему объекты Promise спроектированы именно так, а не иначе. Значение удовлетворенного объекта Promise похоже на возвращаемое значение синхронной функции. Значение отклоненного объекта Promise подобно значению исключения, сгенерированного синхронной функцией. Последнее сходство явно выражено в имени метода .catch(). Ключевые слова async и await берут эффективный код, основанный на Promise, и скрывают объекты Promise, чтобы асинхронный код можно было так же легко читать и понимать, как неэффективный, блокирующий синхронный код.

13.3.1. Выражения await

Ключевое слово await берет объект Promise и превращает его обратно в возвращаемое значение или сгенерированное исключение. Для объекта Promise по имени p выражение await p ожидает до тех пор, пока p не будет урегулирован. Если p удовлетворяется, тогда значением await p будет значение удовлетворения p. С другой стороны, если p отклоняется, тогда выражение await p генерирует значение отклонения p. Обычно мы не применяем ключевое слово await с переменной, которая хранит объект Promise; взамен мы используем его перед вызовом функции, возвращающей объект Promise:

```
let response = await fetch("/api/user/profile");
let profile = await response.json();
```

Критически важно прямо сейчас понять, что ключевое слово await не заставляет вашу программу блокироваться и буквально ничего не делать до тех пор, пока указанный объект Promise не будет урегулирован. Код остается асинхронным и await просто маскирует данный факт. Это означает, что *любой код, в котором применяется await, сам является асинхронным.*

13.3.2. Функции async

Из-за того, что любой код, в котором используется await, становится асинхронным, есть одно важное правило: *вы можете применять ключевое слово await только внутри функций, которые были объявлены с ключевым словом async.*

Вот версия функции getHighScore(), рассмотренной ранее в главе, где задействованы ключевые слова async и await:

```
async function getHighScore() {
    let response = await fetch("/api/user/profile");
    let profile = await response.json();
    return profile.highScore;
}
```

Объявление функции async означает, что возвращаемое значение функции будет объектом Promise, даже если в теле функции не содержится код, связанный с Promise. Если функция async выглядит делающей нормальный возврат, то объект Promise, который представляет собой реальное возвращаемое значение функции, будет разрешен с таким явным возвращаемым значением. Если же функция async выглядит генерирующей исключение, тогда объект Promise, который она возвращает, будет отклонен с этим исключением.

Функция getHighScore() объявлена async и потому она возвращает объект Promise. А поскольку она возвращает объект Promise, мы можем использовать с ней ключевое слово await:

```
displayHighScore(await getHighScore());
```

Не забывайте, что приведенная выше строка кода будет работать, только если она находится внутри еще одной функции async! Вы можете вкладывать выражения await внутри функций async настолько глубоко, насколько хотите. Но если вы находитесь на верхнем уровне[2] или внутри функции, которая по какой-то причине не async, то не сможете применять await и будете вынуждены работать с возвращаемым объектом Promise обычным образом:

```
getHighScore().then(displayHighScore).catch(console.error);
```

Вы можете использовать ключевое слово async с функцией любого вида. Оно работает с ключевым словом function как оператор или как выражение. Оно работает со стрелочными функциями и с сокращенной формой записи методов в классах и объектных литералах. (Различные способы написания функций раскрывались в главе 8.)

13.3.3. Ожидание множества объектов Promise

Предположим, что мы написали свою функцию getJSON() с применением async:

```
async function getJSON(url) {
    let response = await fetch(url);
    let body = await response.json();
    return body;
}
```

[2] Как правило, вы будете использовать await на верхнем уровне в консоли разработчика браузера. И существует ожидающее решения предложение сделать возможным применение ключевого слова await на верхнем уровне в будущей версии JavaScript.

Пусть мы хотим извлечь с помощью этой функции два значения JSON:

```
let value1 = await getJSON(url1);
let value2 = await getJSON(url2);
```

Проблема с таким кодом связана с тем, что он излишне последовательный: извлечение второго URL не начнется до тех пор, пока не завершится извлечение первого URL. Если второй URL не зависит от значения, полученного из первого URL, тогда вероятно мы должны попробовать извлечь два значения одновременно. Этот тот случай, когда проявляется основанная на `Promise` природа функций `async`. Для ожидания множества параллельно выполняющихся функций `async` мы используем `Promise.all()`, как если бы имели дело с объектами `Promise` напрямую:

```
let [value1,value2] = await Promise.all([getJSON(url1),getJSON(url2)]);
```

13.3.4. Детали реализации

Наконец, чтобы понять, как работают функции `async`, имеет смысл подумать о том, что происходит "за кулисами".

Предположим, что вы написали функцию `async` следующего вида:

```
async function f(x) { /* тело */ }
```

Вы можете считать ее функцией, возвращающей `Promise`, которая является оболочкой для тела вашей первоначальной функции:

```
function f(x) {
   return new Promise(function(resolve, reject) {
      try {
         resolve((function(x) { /* body */ })(x));
      }
      catch(e) {
         reject(e);
      }
   });
}
```

Ключевое слово `await` труднее выразить в терминах синтаксической трансформации подобного рода. Но думайте о ключевом слове `await` как о маркере, который разбивает тело функции на отдельные синхронные порции. Интерпретатор ES2017 способен разбивать тело функции на последовательность отдельных подфункций, каждая из которых передается методу `then()` предшествующего объекта `Promise`, помеченного посредством ключевого слова `await`.

13.4. Асинхронная итерация

Мы начинали главу с обсуждения асинхронности, основанной на обратных вызовах и событиях, и когда представляли объекты `Promise`, то отмечали, что они были полезны для одноразовых асинхронных вычислений, но не подходили для применения с источниками повторяющихся асинхронных событий, таки-

ми как setInterval(), событие щелчка в веб-браузере или событие данных в потоке Node. Из-за того, что одиночные объекты Promise не работают для последовательностей асинхронных событий, мы также не можем использовать для них обыкновенные функции async и операторы await.

Однако в ES2018 было предложено решение. Асинхронные итераторы похожи на итераторы, описанные в главе 12, но они основаны на Promise и предназначены для применения с новой формой цикла for/of: for/await.

13.4.1. Цикл for/await

Среда Node 12 делает свои допускающие чтение потоки данных асинхронно итерируемыми. Это означает, что вы можете читать последовательные порции данных из потока с помощью цикла for/await вроде приведенного ниже:

```
const fs = require("fs");

async function parseFile(filename) {
  let stream = fs.createReadStream(filename, { encoding: "utf-8"});
  for await (let chunk of stream) {
    parseChunk(chunk);  // Функция parseChunk() определена
                        // где-то в другом месте
  }
}
```

Как и обычные выражения await, цикл for/await основан на Promise. Грубо говоря, асинхронный итератор производит объект Promise и цикл for/await ожидает, когда этот объект Promise будет удовлетворен, присваивает значение удовлетворения переменной цикла и выполняет тело цикла. Затем он начинает заново, получая еще один объект Promise от итератора и ожидая, пока новый объект Promise будет удовлетворен.

Предположим, у вас есть массив URL:

```
const urls = [url1, url2, url3];
```

Вы можете вызвать fetch() с каждым URL, чтобы получить массив объектов Promise:

```
const promises = urls.map(url => fetch(url));
```

Ранее в главе было показано, что теперь мы могли бы использовать Promise.all() для ожидания, пока все объекты Promise в массиве будут удовлетворены. Но допустим, что нам нужны результаты первой выборки, как только они станут доступными, и мы не хотим ожидать извлечения всех URL. (Разумеется, первое извлечение может занять больше времени, чем все остальные, так что итоговый код не обязательно окажется быстрее, чем в случае применения Promise.all().) Массивы итерируемы, поэтому мы можем проходить по массиву объектов Promise с помощью обыкновенного цикла for/of:

```
for(const promise of promises) {
  response = await promise;
  handle(response);
}
```

В примере кода используется обычный цикл for/of с обычным итератором. Но поскольку данный итератор возвращает объекты Promise, мы также можем применять новый цикл for/await, чтобы слегка упростить код:

```
for await (const response of promises) {
   handle(response);
}
```

В этом случае for/await всего лишь встраивает вызов await в цикл и делает код немного компактнее, но два примера решают одну и ту же задачу. Важно отметить, что оба примера кода работают, только если находятся внутри функций, объявленных как async; таким образом, цикл for/await ничем не отличается от обыкновенного выражения await.

Тем не менее, важно понимать, что в приведенном примере мы используем for/await с обычным итератором. Все становится более интересным, когда речь идет о полностью асинхронных итераторах.

13.4.2. Асинхронные итераторы

Давайте вспомним ряд терминов из главы 12. *Итерируемый* объект — это такой объект, который можно задействовать в цикле for/of. В нем определен метод с символьным именем Symbol.iterator, возвращающий объект *итератора*. Объект итератора имеет метод next(), который можно многократно вызывать для получения значений итерируемого объекта. Метод next() объекта итератора возвращает объекты *результатов итераций*. Объект результата итерации имеет свойство value и/или свойство done.

Асинхронные итераторы похожи на обыкновенные итераторы, но обладают двумя важными отличиями. Во-первых, в асинхронно итерируемом объекте реализован метод с символьным именем Symbol.asyncIterator, а не Symbol.iterator. (Как упоминалось ранее, цикл for/await совместим с обычными итерируемыми объектами, но предпочитает асинхронно итерируемые объекты и опробует сначала метод Symbol.asyncIterator и только затем метод Symbol.iterator.) Во-вторых, метод next() асинхронного итератора возвращает не непосредственно объект результата итерации, а объект Promise, который разрешается в объект результата итерации.

Когда мы применяли for/await с обыкновенным синхронно итерируемым массивом объектов Promise в предыдущем разделе, то работали с синхронными объектами результатов итерации, в которых свойство value хранило объект Promise, но свойство done было синхронным. Подлинные асинхронные итераторы возвращают объекты Promise для объектов результатов итерации и свойства value и done являются асинхронными. Отличие тонкое: в случае асинхронных итераторов выбор момента окончания итерации может делаться асинхронно.

13.4.3. Асинхронные генераторы

Как было показано в главе 12, часто самым легким способом реализации итератора оказывается использование генератора. То же самое справедливо в отношении асинхронных итераторов, которые мы можем реализовать с помощью генераторных функций, объявляемых как async. Асинхронный генератор обладает возможностями асинхронных функций и генераторов: вы можете применять await, как делали бы в обычной асинхронной функции, а также использовать yield, как поступали бы в обыкновенном генераторе. Но выдаваемые посредством yield значения автоматически помещаются в оболочки объектов Promise. Даже синтаксис для асинхронных генераторов является комбинацией: async function и function * объединяются в async function *. Ниже приведен пример, который демонстрирует, каким образом можно было бы применить асинхронный генератор и цикл for/await для повторяющегося выполнения кода через фиксированные интервалы, используя синтаксис цикла вместо функции обратного вызова setInterval():

```
// Оболочка на основе Promise вокруг setTimeout(),
// с которой можно использовать await.
// Возвращает объект Promise, который удовлетворяется
// с указанным количеством миллисекунд.
function elapsedTime(ms) {
   return new Promise(resolve => setTimeout(resolve, ms));
}

// Асинхронная генераторная функция, которая инкрементирует счетчик
// и выдает его указанное (или бесконечное) количество раз
// через указанные интервалы.
async function* clock(interval, max=Infinity) {
   for(let count = 1; count <= max; count++) {// Обыкновенный цикл for
      await elapsedTime(interval);            // Ожидать в течение
                                              // указанного времени.
      yield count;                            // Выдать счетчик.
   }
}

// Тестовая функция, которая использует асинхронный генератор
// с циклом for/await.
async function test() { //Асинхронная, поэтому можно применять for/await.
   for await (let tick of clock(300, 100)) { // Цикл 100 раз каждые
                                             // 300 миллисекунд.
      console.log(tick);
   }
}
```

13.4.4. Реализация асинхронных итераторов

Вместо применения асинхронных генераторов для реализации асинхронных итераторов их также можно реализовать напрямую за счет определения объекта с методом Symbol.asyncIterator(), возвращающим объект с методом next(), который возвращает объект Promise, разрешаемый в объект результа-

та итерации. В показанном далее коде мы заново реализуем функцию clock() из предыдущего примера, чтобы она не была генератором и взамен просто возвращала асинхронно итерируемый объект. Обратите внимание, что метод next() в текущем примере не возвращает явно объект Promise; вместо этого мы всего лишь объявляем next() асинхронным:

```
function clock(interval, max=Infinity) {
   // Версия setTimeout с Promise, с которой можно использовать await.
   // Обратите внимание, что функция принимает абсолютное время,
   // а не интервал.
   function until(time) {
     return new Promise(resolve=>setTimeout(resolve,time-Date.now()));
   }

   // Возвратить асинхронно итерируемый объект.
   return {
      startTime: Date.now(),    // Запомнить, когда мы начали.
      count: 1,                 // Запомнить, на какой мы итерации.
      async next() {            // Метод next() делает это итератором.
         if (this.count > max) {     // Мы закончили?
            return { done: true };  // Результат итерации готов.
         }
         // Выяснить, когда должна начаться следующая итерация,
         let targetTime = this.startTime + this.count * interval;
         // ожидать в течение этого времени
         await until(targetTime);
         // и возвратить значение счетчика в объекте результата итерации
         return { value: this.count++ };
      },
  // Этот метод означает, что данный объект итератора тоже итерируемый.
      [Symbol.asyncIterator]() { return this; }
   };
}
```

В версии функции clock() на основе итератора ликвидирован недостаток, имеющийся в версии на основе генератора. Как видите, в новом коде мы нацелены на абсолютное время, когда должна начинаться каждая итерация, и вычитаем из него текущее время, чтобы рассчитать интервал, который передается setTimeout(). Если использовать clock() с циклом for/await, то эта версия будет осуществлять итерации в цикле более точно через указанный интервал, поскольку она учитывает время, требующееся для действительного выполнения тела цикла. Причем исправление касается не только точности расчета времени. Цикл for/await всегда ожидает, пока объект Promise, возвращенный одной итерацией, не будет удовлетворен, прежде чем начинать следующую итерацию. Но если вы применяете асинхронный итератор без цикла for/await, то ничего не помешает вам вызывать метод next() в любой желаемый момент. В версии clock(), основанной на генераторе, если вы вызовите метод next() три раза подряд, то получите три объекта Promise, которые будут все удовлетворены почти одновременно, и вероятно это не то, что вам нужно. В реализованной здесь версии clock() на основе итератора такая проблема не возникает.

Преимущество асинхронных итераторов в том, что они позволяют нам представлять потоки асинхронных событий или данных. Обсуждаемая ранее функция `clock()` оказалась довольно простой в написании, потому что источником асинхронности были вызовы `setTimeout()`, которые мы делали самостоятельно. Но когда мы пытаемся работать с другими асинхронными источниками, такими как запуск обработчиков событий, реализовать асинхронные итераторы становится значительно труднее. Дело в том, что у нас обычно есть одна функция обработки событий, реагирующая на события, но каждый вызов метода `next()` итератора обязан возвращать отдельный объект `Promise`, и до того, как первый объект `Promise` будет разрешен, может случиться много вызовов `next()`. Это означает, что любой метод асинхронного итератора должен быть в состоянии поддерживать внутреннюю очередь объектов `Promise`, которые он разрешает в порядке возникновения асинхронных событий. Если мы инкапсулируем такое поведение с очередью объектов `Promise` в виде класса `AsyncQueue`, тогда будет намного легче писать асинхронные итераторы на основе `AsyncQueue`[3].

В показанном ниже классе `AsyncQueue` имеются методы `enqueue()` и `dequeue()`, что вполне ожидаемо для класса очереди. Однако метод `dequeue()` возвращает не фактическое значение, а объект `Promise`, т.е. метод `dequeue()` можно вызывать до того, как `enqueue()` когда-либо будет вызван. Класс `AsyncQueue` также является асинхронным итератором и предназначен для использования с циклом `for/await`, тело которого выполняется однократно каждый раз, когда новое значение асинхронно помещается в очередь. (Класс `AsyncQueue` имеет метод `close()`. После его вызова помещать дополнительные значения в очередь нельзя. Когда закрытая очередь пуста, цикл `for/await` останавливает свое выполнение.)

Обратите внимание, что реализация `AsyncQueue` не задействует `async` или `await`, взамен работая напрямую с объектами `Promise`. Код несколько сложен, и вы можете проработать его, чтобы проверить свое понимание материала, который был раскрыт в этой длинной главе. Даже если вы не полностью поймете реализацию `AsyncQueue`, то ознакомьтесь с коротким примером, который приведен сразу за ней: в нем реализован простой, но очень интересный асинхронный итератор поверх класса `AsyncQueue`.

```
/**
 * Класс асинхронно итерируемой очереди. Добавляйте значения
 * с помощью enqueue() и удаляйте их с помощью dequeue().
 * Метод dequeue() возвращает объект Promise, поэтому значения
 * можно извлекать из очереди до того, как они будут в неепомещены.
 * Класс реализует методы [Symbol.asyncIterator] и next(), так что
 * его можно использовать с циклом for/await (который не завершится
 * до тех пор, пока не будет вызван метод close().)
 */
class AsyncQueue {
  constructor() {
```

[3] Я узнал об этом подходе к асинхронной итерации из блога доктора Акселя Раушмайера, https://2ality.com/.

```
    // Здесь хранятся значения, которые были помещены в очередь,
    // но еще не извлечены.
    this.values = [];

    // Когда объекты Promise извлекаются до того, как
    // соответствующие им значения помещаются в очередь, здесь
    // сохраняются методы разрешения для таких объектов Promise.
    this.resolvers = [];

    //После закрытия дополнительные значения помещать в очередь нельзя,
    //и неудовлетворенные объекты Promise больше не возвращаются.
    this.closed = false;
}
enqueue(value) {
    if (this.closed) {
        throw new Error("AsyncQueue closed");
                    // Очередь AsyncQueue закрыта
    }
    if (this.resolvers.length > 0) {
        // Если это значение уже снабжалось объектом Promise,
        // тогда разрешить данный объект Promise.
        const resolve = this.resolvers.shift();
        resolve(value);
    }
    else {
        // В противном случае поместить значение в очередь.
        this.values.push(value);
    }
}
dequeue() {
    if (this.values.length > 0) {
        // Если есть значение, помещенное в очередь, тогда
        // возвратить для него разрешенный объект Promise.
        const value = this.values.shift();
        return Promise.resolve(value);
    }
    else if (this.closed) {
    // Если значения в очереди отсутствуют и очередь закрыта, тогда
    // возвратить разрешенный объект Promise для маркера конца потока
        return Promise.resolve(AsyncQueue.EOS);
    }
    else {
   // В противном случае возвратить неразрешенный объект Promise,
   // поместив в очередь функцию разрешения для будущего использования
        return new Promise((resolve) =>
                { this.resolvers.push(resolve); });
    }
}
```

```
    close() {
    // После закрытия дополнительные значения помещаться в очередь не будут.
    // Таким образом, разрешить любые ожидающие решения объекты Promise
    // в маркер конца потока.
        while(this.resolvers.length > 0) {
            this.resolvers.shift()(AsyncQueue.EOS);
        }
        this.closed = true;
    }

    // Определение метода, который делает этот класс
    // асинхронно итерируемым.
    [Symbol.asyncIterator]() { return this; }

    // Определение метода, который делает это асинхронным итератором.
    // Объект Promise, возвращенный методом dequeue(), разрешается
    // в значение или сигнал EOS, если очередь закрыта.
    // Здесь нам необходимо возвратить объект Promise,
    // который разрешается в объект результата итерации.
    next() {
        return this.dequeue().then(value => (value === AsyncQueue.EOS)
            ? { value: undefined, done: true }
            : { value: value, done: false });
    }
}

// Сигнальное значение, возвращаемое методом dequeue()
// для пометки конца потока, когда очередь закрыта.
AsyncQueue.EOS = Symbol("end-of-stream");
```

Поскольку в классе AsyncQueue определены основы асинхронной итерации, мы можем создавать собственные более интересные асинхронные итераторы, просто помещая значения в очередь асинхронным образом. Ниже показан пример применения класса AsyncQueue для вырабатывания потока событий веб-браузера, которые могут быть обработаны с помощью цикла for/await:

```
// Помещает события указанного типа, относящиеся к указанному
// элементу документа, в объект AsyncQueue и возвращает очередь
// для использования в качестве потока событий.
function eventStream(elt, type) {
    const q = new AsyncQueue();                    // Создать очередь.
    elt.addEventListener(type, e=>q.enqueue(e));  // Поместить события
                                                   // в очередь.
    return q;
}

async function handleKeys() {
    // Получить поток событий нажатия клавиш и пройти по ним в цикле.
    for await (const event of eventStream(document, "keypress")) {
        console.log(event.key);
    }
}
```

13.5. Резюме

Ниже перечислены основные моменты, которые рассматривались в главе.

- В реальности программирование на JavaScript по большей части является асинхронным.
- Асинхронность традиционно обрабатывалась с помощью событий и функций обратного вызова. Тем не менее, это может стать сложным, потому что в итоге приходится иметь дело с множеством уровней обратных вызовов, вложенных внутрь других обратных вызовов, а также из-за того, что трудно обеспечить надежную обработку ошибок.
- Объекты `Promise` предлагают новый способ структурирования функций обратного вызова. При корректном применении (к сожалению, объекты `Promise` легко использовать некорректно) они способны преобразовывать асинхронный код, который был вложенным, в линейные цепочки вызовов `then()`, где один асинхронный шаг вычисления следует за другим. Кроме того, объекты `Promise` позволяют централизовать код обработки ошибок внутри единственного вызова `catch()` в конце цепочки вызовов `then()`.
- Ключевые слова `async` и `await` позволяют писать асинхронный код, который внутренне основан на объектах `Promise`, но выглядит как синхронный код. В итоге код легче читать и понимать. Если функция объявлена с ключевым словом `async`, тогда она неявно возвращает объект `Promise`. Внутри функции `async` можно ожидать с помощью `await` объект `Promise` (или функцию, которая возвращает `Promise`), как если бы значение `Promise` вычислялось синхронным образом.
- Объекты, которые являются асинхронно итерируемыми, могут использоваться с циклом `for/await`. Асинхронно итерируемые объекты создаются за счет реализации метода `[Symbol.asyncIterator]()` или путем вызова генераторной функции `async function *`. Асинхронные итераторы предлагают альтернативу событиям "данных" в потоках среды Node и могут применяться для представления потока входных событий от пользователя в коде JavaScript стороны клиента.

ГЛАВА 14

Метапрограммирование

В этой главе рассматривается несколько расширенных возможностей JavaScript, которые обычно не применяются при повседневном программировании, но могут быть полезны программистам, пишущим многократно используемые библиотеки, и интересны тем, кто желает повозиться с деталями того, как ведут себя объекты JavaScript.

Многие из описанных здесь средств можно условно назвать "метапрограммированием": если нормальное программирование предусматривает написание кода для манипулирования данными, то метапрограммирование предполагает написание кода для манипулирования другим кодом. В динамическом языке вроде JavaScript границы между программированием и метапрограммированием размыты — программисты, привыкшие к более статическим языкам, могут относить к метапрограммированию даже простую возможность прохода по свойствам объекта с помощью цикла `for/in`.

Темы метапрограммирования, раскрываемые в главе, включают:

- управление возможностями перечисления, делегирования и конфигурирования свойств объекта (раздел 14.1);
- управление расширяемостью объектов и создание "запечатанных" и "замороженных" объектов (раздел 14.2);
- запрашивание и установка прототипов объектов (раздел 14.3);
- точная настройка поведения ваших типов посредством хорошо известных объектов `Symbol` (раздел 14.4);
- создание проблемно-ориентированных языков с помощью шаблонных теговых функций (раздел 14.5);
- зондирование объектов посредством методов `reflect` (раздел 14.6);
- управление поведением объектов с помощью класса `Proxy` (раздел 14.7).

14.1. Атрибуты свойств

Свойства объекта JavaScript, конечно, имеют имена и значения, но каждое свойство также имеет три ассоциированных атрибута, которые указывают, как свойство себя ведет и что с ним можно делать:

- атрибут `writable` (записываемое) указывает, можно ли изменять значение свойства;
- атрибут `enumerable` (перечислимое) указывает, перечисляется ли свойство посредством цикла `for/in` и метода `Object.keys()`;
- атрибут `configurable` (конфигурируемое) указывает, может ли свойство быть удалено, а также можно ли изменять атрибуты свойства.

Свойства, определяемые в объектных литералах или обыкновенным присваиванием объекту, будут записываемыми, перечислимыми и конфигурируемыми. Но многие свойства, определенные в стандартной библиотеке JavaScript, таковыми не являются. В настоящем разделе объясняется API-интерфейс для запрашивания и установки атрибутов свойств. Этот API-интерфейс особенно важен для авторов библиотек по следующим причинам:

- он позволяет им добавлять методы к объектам прототипов и делать их неперечислимыми подобно встроенным методам;
- он позволяет им "запирать" свои объекты, определяя свойства, которые нельзя изменять или удалять.

Вспомните из подраздела 6.10.6, что в то время как "свойства данных" имеют значение, "свойства с методами доступа" взамен имеют метод получения и/или метод установки. Для целей текущего раздела мы собираемся относить методы получения и установки свойства с методами доступа к атрибутам свойства. Следуя такой логике, мы даже будем полагать, что значение свойства данных тоже является атрибутом. Соответственно, мы можем заявить, что свойство имеет имя и четыре атрибута. Четыре атрибута свойства данных — это `value`, `writable`, `enumerable` и `configurable`. Свойства с методами доступа не имеют атрибута `value` или атрибута `writable`: возможность записи в них определяется наличием или отсутствием метода установки. Таким образом, четыре атрибута свойства с методами доступа — это `get`, `set`, `enumerable` и `configurable`.

Методы JavaScript запрашивания и установки атрибутов свойства для представления набора из четырех атрибутов применяют объект, который называется *дескриптором свойства*. Объект дескриптора свойства имеет свойства с теми же именами, что и атрибуты свойства, которое он описывает. Следовательно, объект дескриптора для свойства данных располагает свойствами `value`, `writable`, `enumerable` и `configurable`. Дескриптор для свойства с методами доступа вместо `value` и `writable` имеет свойства `get` и `set`. Свойства `writable`, `enumerable` и `configurable` являются булевскими значениями, а свойства `get` и `set` — значениями типа функций.

Чтобы получить дескриптор свойства для именованного свойства указанного объекта, необходимо вызвать метод `Object.getOwnPropertyDescriptor()`:

```
// Возвращается {value: 1, writable:true, enumerable:true,
//               configurable:true}
Object.getOwnPropertyDescriptor({x: 1}, "x");

// Объект со свойством с методами доступа только для чтения.
const random = {
   get octet() { return Math.floor(Math.random()*256); },
};
// Возвращается { get: /*func*/, set:undefined, enumerable:true,
//                configurable:true}
Object.getOwnPropertyDescriptor(random, "octet");

// Возвращается undefined для унаследованных и несуществующих свойств.
Object.getOwnPropertyDescriptor({}, "x") // => undefined;
                                         // несуществующее свойство
Object.getOwnPropertyDescriptor({}, "toString")  // => undefined;
                                         // унаследованное свойство
```

Как подразумевает его имя, метод `Object.getOwnPropertyDescriptor()` работает только с собственными свойствами. Для запрашивания атрибутов унаследованных свойств придется явно обходить цепочку прототипов. (См. `Object.getPrototypeOf()` в разделе 14.3, а также похожую функцию `Reflect.getOwnPropertyDescriptor()` в разделе 14.6.)

Чтобы установить атрибуты свойства или создать новое свойство с заданными атрибутами, нужно вызывать метод `Object.defineProperty()`, передав ему модифицируемый объект, имя создаваемого либо изменяемого свойства и объект дескриптора свойства:

```
let o = {};         // Начать с объекта, вообще не имеющего свойств.
// Добавить неперечислимое свойство данных x со значением 1.
Object.defineProperty(o, "x", {
   value: 1,
   writable: true,
   enumerable: false,
   configurable: true
});
// Проверить, что свойство присутствует, но не перечисляется.
o.x                 // => 1
Object.keys(o)      // => []

// Модифицировать свойство x, чтобы оно допускало только чтение.
Object.defineProperty(o, "x", { writable: false });

// Попытаться изменить значение свойства.
o.x = 2;            // Молча терпит неудачу или генерирует TypeError
                    // в строгом режиме.
o.x                 // => 1

// Свойство по-прежнему конфигурируемое, поэтому
// мы можем изменить его значение следующим образом:
Object.defineProperty(o, "x", { value: 2 });
o.x                 // => 2
// Превратить свойство данных x в свойство с методами доступа.
Object.defineProperty(o, "x", { get: function() { return 0; } });
o.x                 // => 0
```

Дескриптор свойства, передаваемый методу `Object.defineProperty()`, не обязан включать все четыре атрибута. В случае создания нового свойства атрибуты, которые не указаны, будут равны `false` или `undefined`. В случае изменения существующего свойства опущенные атрибуты просто остаются неизменными. Обратите внимание, что метод `Object.defineProperty()` изменяет существующее собственное свойство или создает новое собственное свойство, но он не будет изменять унаследованное свойство. В разделе 14.6 описана очень похожая функция `Reflect.defineProperty()`.

Если необходимо создать или модифицировать более одного свойства за раз, тогда следует использовать метод `Object.defineProperties()`. В первом аргументе передается объект, подлежащий модификации, а во втором — объект, который отображает имена создаваемых или изменяемых свойств на соответствующие дескрипторы свойств, например:

```
let p = Object.defineProperties({}, {
  x: { value: 1,writable: true,enumerable: true,configurable: true },
  y: { value: 1,writable: true,enumerable: true,configurable: true },
  r: {
      get() { return Math.sqrt(this.x*this.x + this.y*this.y); },
      enumerable: true,
      configurable: true
  }
});
p.r                 // => Math.SQRT2
```

Приведенный выше код начинает с пустого объекта и затем добавляет к нему два свойства данных и одно свойство с методами доступа, допускающее только чтение. Он полагается на тот факт, что `Object.defineProperties()` возвращает модифицированный объект (как делает `Object.defineProperty()`).

Метод `Object.create()` был представлен в разделе 6.2. Там объяснялось, что в первом аргументе этого метода передается объект-прототип для создаваемого объекта. Метод также принимает второй необязательный аргумент, который совпадает со вторым аргументом метода `Object.defineProperties()`. Если передать методу `Object.create()` набор дескрипторов свойств, тогда они будут применяться для добавления свойств во вновь созданный объект.

Методы `Object.defineProperty()` и `Object.defineProperties()` генерируют исключение `TypeError`, если создание или изменение свойства не разрешено. Так случается при попытке добавить новое свойство к нерасширяемому (см. раздел 14.2) объекту. Другие причины, по которым упомянутые методы могут генерировать `TypeError`, связаны с самими атрибутами. Атрибут `writable` управляет возможностью изменения атрибута `value`. Атрибут `configurable` управляет возможностью изменения остальных атрибутов (и также указывает, можно ли удалять свойство). Однако правила не настолько прямолинейны. Скажем, можно изменить значение свойства, не допускающего запись, если оно конфигурируемое. Кроме того, записываемое свойство можно сделать не допускающим запись, даже когда оно не является конфигурируемым. Полные правила описаны ниже.

Вызовы `Object.defineProperty()` или `Object.defineProperties()` генерируют `TypeError` в случае нарушения следующих правил.

- Если объект не является расширяемым, тогда вы можете редактировать его существующие собственные свойства, но не добавлять к нему новые свойства.
- Если свойство не является конфигурируемым, тогда вы не можете изменять его атрибуты `configurable` или `enumerable`.
- Если свойство с методами доступа не является конфигурируемым, тогда вы не можете изменять его метод получения или установки, равно как не можете превращать его в свойство данных.
- Если свойство данных не является конфигурируемым, тогда вы не можете превращать его в свойство с методами доступа.
- Если свойство данных не является конфигурируемым, тогда вы не можете изменять его атрибут `writable` с `false` на `true`, но можете изменять его с `true` на `false`.
- Если свойство данных не является конфигурируемым и записываемым, тогда вы не можете изменять его значение. Тем не менее, вы можете изменять значение свойства, которое допускает конфигурирование, но не запись (т.к. это то же, что сделать его записываемым, затем изменить значение и преобразовать обратно в не допускающее запись).

В разделе 6.7 была описана функция `Object.assign()`, которая копирует значения свойств из одного или большего количества исходных объектов в целевой объект. Функция `Object.assign()` копирует только перечислимые свойства и значения свойств, но не атрибуты свойств. Обычно именно такое действие и желательно, но это означает, что если, скажем, один из исходных объектов имеет свойство с методами доступа, то в целевой объект скопируется значение, возвращенное методом получения, а не сам метод получения. В примере 14.1 показано, как можно использовать `Object.getOwnPropertyDescriptor()` и `Object.defineProperty()` для создания версии функции `Object.assign()`, которая копирует целые дескрипторы свойств, а не только значения свойств.

Пример 14.1. Копирование свойств и их атрибутов из одного объекта в другой

```
/*
 * Определение новой функции Object.assignDescriptors(), которая
 * работает подобно Object.assign(), но копирует в целевой объект
 * из исходных объектов дескрипторы свойств, а не только значения свойств.
 * Эта функция копирует все собственные свойства, как перечислимые,
 * так и неперечислимые. И поскольку копируются дескрипторы, то она
 * копирует функции получения из исходных объектов и переписывает
 * функции установки в целевом объекте, а не только вызывает
 * эти функции получения и установки.
 *
```

```
 * Object.assignDescriptors() распространяет любые исключения TypeError,
 * сгенерированные Object.defineProperty(). Они могут произойти, если
 * целевой объект является запечатанным или замороженным либо если
 * любое из свойств исходных объектов пытается изменить существующее
 * неконфигурируемое свойство целевого объекта.
 *
 * Обратите внимание, что свойство assignDescriptors добавляется
 * в Object с помощью Object.defineProperty(), поэтому новую функцию
 * можно создавать как неперечислимое свойство вроде Object.assign().
 */
Object.defineProperty(Object, "assignDescriptors", {
    // Соответствует атрибутам Object.assign().
    writable: true,
    enumerable: false,
    configurable: true,
    // Функция, которая является значением свойства assignDescriptors.
    value: function(target, ...sources) {
        for(let source of sources) {
            for(let name of Object.getOwnPropertyNames(source)) {
                let desc = Object.getOwnPropertyDescriptor(source, name);
                Object.defineProperty(target, name, desc);
            }

            for(let symbol of Object.getOwnPropertySymbols(source)) {
                let desc = Object.getOwnPropertyDescriptor(source, symbol);
                Object.defineProperty(target, symbol, desc);
            }
        }
        return target;
    }
});

let o = {c: 1, get count() {return this.c++;}}; // Определить объект с
                                                // методом получения.
let p = Object.assign({}, o);           // Копировать значения свойств.
let q = Object.assignDescriptors({}, o); // Копировать дескрипторы свойств.
p.count      // => 1: Теперь это просто свойство данных, поэтому
p.count      // => 1: ...счетчик не инкрементируется.
q.count      // => 2: Инкрементировался однократно,
             //       когда мы скопировали его в первый раз,
q.count      // => 3: ...но мы скопировали метод получения,
             //       так что счетчик инкрементируется.
```

14.2. Расширяемость объектов

Атрибут extensible (расшияремый) объекта указывает, можно ли добавлять к объекту новые свойства. Обычные объекты JavaScript по умолчанию расширяемы, но вы можете изменять это с помощью функций, описанных в настоящем разделе.

Чтобы определить, является ли объект расширяемым, его необходимо передать функции `Object.isExtensible()`. Чтобы сделать объект нерасширяемым, его нужно передать функции `Object.preventExtensions()`. Затем любая попытка добавления нового свойства к объекту будет приводить к генерации исключения `TypeError` в строгом режиме и просто к неудаче в нестрогом режиме. Кроме того, попытка изменения прототипа (см. раздел 14.3) нерасширяемого объекта всегда будет вызывать генерацию `TypeError`.

Обратите внимание, что после того, как вы сделали объект нерасширяемым, не существует способа сделать его снова расширяемым. Также имейте в виду, что вызов `Object.preventExtensions()` оказывает воздействие только на расширяемость самого объекта. Если к прототипу нерасширяемого объекта добавляются новые свойства, то они будут унаследованы нерасширяемым объектом.

В разделе 14.6 описаны две похожие функции, `Reflect.isExtensible()` и `Reflect.preventExtensions()`.

Атрибут `extensible` предназначен для того, чтобы иметь возможность "запирать" объекты в известном состоянии и предотвращать внешнее вмешательство. Атрибут `extensible` объектов часто применяется в сочетании с атрибутами `configurable` и `writable` свойств, а в JavaScript определены функции, которые облегчают установку этих атрибутов вместе.

- Функция `Object.seal()` работает подобно `Object.preventExtensions()`, но в дополнение к тому, что делает объект нерасширяемым, она также превращает все собственные свойства объекта в неконфигурируемые. Это означает, что к объекту нельзя добавлять новые свойства, а существующие свойства не удастся удалить или конфигурировать. Однако существующие свойства, которые являются записываемыми, по-прежнему можно устанавливать. Способа распечатать запечатанный объект не предусмотрено. Выяснить, запечатан ли объект, можно посредством функции `Object.isSealed()`.
- Функция `Object.freeze()` запирает объект еще сильнее. Помимо того, что она делает объект нерасширяемым и его свойства неконфигурируемыми, `Object.freeze()` также превращает все собственные свойства данных объекта в допускающие только чтение. (Если объект имеет свойства с методами установки, то они не затрагиваются и по-прежнему могут вызываться операциями присваивания свойствам.) Выяснить, заморожен ли объект, можно с помощью функции `Object.isFrozen()`.

Важно понимать, что функции `Object.seal()` и `Object.freeze()` воздействуют только на переданный им объект: они не оказывают никакого влияния на прототип этого объекта. Если вас интересует полное запирание объекта, то вероятно вам понадобится запечатать или заморозить также объекты в цепочке прототипов.

Функции `Object.preventExtensions()`, `Object.seal()` и `Object.freeze()` возвращают объект, который им передавался, и потому их можно использовать во вложенных вызовах:

```
// Создать запечатанный объект с замороженным прототипом
// и неперечислимым свойством.
let o = Object.seal(Object.create(Object.freeze({x: 1}),
                                  {y: {value: 2, writable: true}}));
```

Если вы разрабатываете библиотеку JavaScript, которая передает объекты в функции обратного вызова, написанные пользователями вашей библиотеки, то можете применить `Object.freeze()` к таким объектам, чтобы предотвратить их модификацию в коде пользователей. Делается это легко и удобно, но существуют компромиссы: например, замороженные объекты могут мешать использованию распространенных стратегий тестирования JavaScript.

14.3. Атрибут `prototype`

Атрибут `prototype` объекта указывает объект, из которого наследуются свойства. (Прототипы и наследование свойств рассматривались в подразделах 6.2.3 и 6.3.2.) Данный атрибут настолько важен, что мы обычно говорим просто "прототип объекта o", а не "атрибут `prototype` объекта o". Вспомните также, что когда слово `prototype` представлено с применением шрифта кода, оно ссылается на обыкновенное свойство объекта, не на атрибут `prototype`: в главе 9 объяснялось, что свойство `prototype` функции конструктора указывает атрибут `prototype` объектов, созданных с помощью этого конструктора.

Атрибут `prototype` устанавливается, когда объект создается. Объекты, созданные из объектных литералов, используют в качестве своих прототипов `Object.prototype`. Объекты, созданные посредством `new`, применяют для своих прототипов значение свойства `prototype` функции конструктора. Объекты, созданные с помощью `Object.create()`, используют в качестве своих прототипов первый аргумент данной функции (который может быть `null`).

Чтобы запросить прототип любого объекта, нужно передать объект функции `Object.getPrototypeOf()`:

```
Object.getPrototypeOf({})          // => Object.prototype
Object.getPrototypeOf([])          // => Array.prototype
Object.getPrototypeOf(()=>{})      // => Function.prototype
```

В разделе 14.6 описана очень похожая функция, `Reflect.getPrototypeOf()`.

Для определения, является ли один объект прототипом (или частью цепочки прототипов) другого объекта, применяется метод `isPrototypeOf()`:

```
let p = {x: 1};                    // Определить объект прототипа.
let o = Object.create(p);          // Создать объект с этим прототипом
p.isPrototypeOf(o)                 // => true: o наследуется из p
Object.prototype.isPrototypeOf(p)  // => true: p наследуется
                                   //         из Object.prototype
Object.prototype.isPrototypeOf(o)  // => true: o тоже
```

Обратите внимание, что `isPrototypeOf()` выполняет функцию, похожую на операцию `instanceof` (см. подраздел 4.9.4).

Атрибут prototype объекта устанавливается, когда объект создается, и по обыкновению остается фиксированным. Тем не менее, с использованием Object.setPrototypeOf() прототип объекта можно изменять:

```
let o = {x: 1};
let p = {y: 2};
Object.setPrototypeOf(o, p);    // Установить прототип объекта o в p.
o.y                             // => 2: o теперь наследует свойство y
let a = [1, 2, 3];
Object.setPrototypeOf(a, p);    // Установить прототип массива a в p
a.join                  // => undefined: a больше не имеет метода join()
```

Обычно необходимость в применении Object.setPrototypeOf() отсутствует. Реализации JavaScript могут проводить энергичную оптимизацию, основываясь на допущении о том, что прототип объекта фиксирован и неизменен. Таким образом, если вы когда-либо вызовите Object.setPrototypeOf(), то любой код, который использует измененные объекты, может выполняться гораздо медленнее, чем в нормальных обстоятельствах.

В разделе 14.6 описана похожая функция, Reflect.setPrototypeOf().

В некоторых ранних браузерных реализациях JavaScript атрибут prototype объекта предоставлялся через свойство __proto__ (содержащее два подчеркивания в начале и в конце). Указанное свойство давно устарело, но достаточно много существующего кода в веб-сети зависит от __proto__ и потому стандарт ECMAScript делает его обязательным для всех реализаций JavaScript, которые выполняются в веб-браузерах. (В Node оно тоже поддерживается, хотя стандарт не требует его для Node.) В современном JavaScript свойство __proto__ допускает чтение и запись, и вы можете (но не должны) применять его как альтернативу использованию функций Object.getPrototypeOf() и Object.setPrototypeOf(). Однако одно интересное применение __proto__ связано с определением прототипа объектного литерала:

```
let p = {z: 3};
let o = {
   x: 1,
   y: 2,
   __proto__: p
};
o.z  // => 3: o унаследован от p
```

14.4. Хорошо известные объекты Symbol

Тип Symbol был добавлен к языку JavaScript в версии ES6, и одной из главных причин его появления было безопасное добавление в язык расширений без нарушения совместимости с кодом, уже развернутым в веб-сети. Пример приводился в главе 12, где мы выясняли, как можно сделать класс итерируемым за счет реализации метода, "именем" которого является Symbol.iterator.

Symbol.iterator — лучше всех известный пример "хорошо известных объектов Symbol". Существует набор значений Symbol, хранящихся в виде свойств

фабричной функции Symbol(), которые используются для того, чтобы позволить коду JavaScript управлять определенными аспектами поведения объектов и классов. В последующих подразделах будут описаны все хорошо известные объекты Symbol и продемонстрированы способы их применения.

14.4.1. Symbol.iterator и Symbol.asyncIterator

Symbol.iterator и Symbol.asyncIterator позволяют объектам или классам делать себя итерируемыми или асинхронно итерируемыми. Они подробно обсуждались в главе 12 и подразделе 13.4.2 соответственно, а здесь упоминаются лишь ради полноты.

14.4.2. Symbol.hasInstance

Во время описания операции instanceof в подразделе 4.9.4 говорилось о том, что с правой стороны должна быть функция конструктора, а выражение o instanceof f вычислялось путем поиска значения f.prototype внутри цепочки прототипов объекта o. Сказанное по-прежнему справедливо, но в ES6 и далее Symbol.hasInstance предлагает альтернативу. В ES6, если с правой стороны instanceof находится любой объект с методом [Symbol.hasInstance], то этот метод вызывается со значением с левой стороны в качестве своего аргумента и возвращаемое значение метода, преобразованное в булевское значение, становится значением операции instanceof. И, разумеется, если значение с правой стороны не имеет метода [Symbol.hasInstance], но является функцией, тогда операция instanceof ведет себя обычным образом.

Symbol.hasInstance означает, что мы можем использовать операцию instanceof для обобщенной проверки типов с соответственно определенными объектами псевдотипов. Вот пример:

```
// Определить объект как "тип", который можно использовать
// с операцией instanceof.
let uint8 = {
    [Symbol.hasInstance](x) {
        return Number.isInteger(x) && x >= 0 && x <= 255;
    }
};
128 instanceof uint8        // => true
256 instanceof uint8        // => false: слишком большой
Math.PI instanceof uint8    // => false: не целый
```

Обратите внимание, что приведенный пример искусный, но сбивающий с толку, поскольку в нем применяется отличающийся от класса объект, где в нормальном случае ожидался бы класс. Было бы столь же легко — и яснее для читателей кода — написать функцию isUint8() вместо того, чтобы полагаться на такое поведение Symbol.hasInstance.

14.4.3. Symbol.toStringTag

Если вы вызовите метод `toString()` базового объекта JavaScript, то получите строку `"[object Object]"`:

```
{}.toString()          // => "[object Object]"
```

Вызвав ту же самую функцию `Object.prototype.toString()` как метод экземпляра встроенных типов, вы получите более интересные результаты:

```
Object.prototype.toString.call([])      // => "[object Array]"
Object.prototype.toString.call(/./)     // => "[object RegExp]"
Object.prototype.toString.call(()=>{}) // => "[object Function]"
Object.prototype.toString.call("")      // => "[object String]"
Object.prototype.toString.call(0)       // => "[object Number]"
Object.prototype.toString.call(false)   // => "[object Boolean]"
```

Оказывается, что вы можете использовать такую методику `Object.prototype.toString().call()` с любым значением JavaScript, чтобы получить "атрибут класса" объекта, содержащий информацию о типе, которая по-другому не доступна. Следующая функция `classof()` возможно полезнее, чем операция typeof, которая не делает никаких различий между типами объектов:

```
function classof(o) {
   return Object.prototype.toString.call(o).slice(8,-1);
}
classof(null)          // => "Null"
classof(undefined)     // => "Undefined"
classof(1)             // => "Number"
classof(10n**100n)     // => "BigInt"
classof("")            // => "String"
classof(false)         // => "Boolean"
classof(Symbol())      // => "Symbol"
classof({})            // => "Object"
classof([])            // => "Array"
classof(/./)           // => "RegExp"
classof(()=>{})        // => "Function"
classof(new Map())     // => "Map"
classof(new Set())     // => "Set"
classof(new Date())    // => "Date"
```

До версии ES6 такое особое поведение метода `Object.prototype.toString()` было доступным только для экземпляров встроенных типов, и вызов этой функции `classof()` с экземпляром класса, который вы определили самостоятельно, возвращал просто `"Object"`. Тем не менее, в ES6 функция `Object.prototype.toString()` ищет в своем аргументе свойство с символьным именем `Symbol.toStringTag` и в случае существования такого свойства применяет в выводе его значение. Это означает, что если вы определяете собственный класс, то можете легко заставить его работать с функциями, подобными `classof()`:

```
class Range {
   get [Symbol.toStringTag]() { return "Range"; }
   // остальной код класса не показан
}
let r = new Range(1, 10);
Object.prototype.toString.call(r)   // => "[object Range]"
classof(r)                          // => "Range"
```

14.4.4. Symbol.species

До версии ES6 в JavaScript не предлагалось ни одного реального способа создания надежных подклассов встроенных классов вроде Array. Однако в ES6 вы можете расширить любой встроенный класс, просто используя ключевые слова class и extends. Прием демонстрировался в подразделе 9.5.2 на простом подклассе класса Array:

```
// Тривиальный подкласс Array, который добавляет методы получения
// для первого и последнего элементов.
class EZArray extends Array {
   get first() { return this[0]; }
   get last() { return this[this.length-1]; }
}
let e = new EZArray(1,2,3);
let f = e.map(x => x * x);
e.last  // => 3: последний элемент массива EZArray по имени e
f.last  // => 9: f - также объект EZArray со свойством last
```

В классе Array определены методы concat(), filter(), map(), slice() и splice(), которые возвращают массивы. Когда мы создаем подкласс массива наподобие EZArray, наследующий указанные методы, экземпляры чего должны возвращать унаследованные методы — Array или EZArray? Можно привести доводы в пользу любого выбора, но спецификация ES6 говорит о том, что пять возвращающих массивы методов будут (по умолчанию) возвращать экземпляры подкласса.

Ниже описано, как все работает.

- В ES6 и последующих версиях конструктор Array() имеет свойство с символьным именем Symbol.species. (Обратите внимание, что этот объект Symbol применяется как имя свойства в функции конструктора. Большинство остальных описанных здесь хорошо известных объектов Symbol используются в качестве имен методов объекта-прототипа.)
- Когда мы создаем подкласс с помощью extends, конструктор результирующего подкласса наследует свойства из конструктора суперкласса. (Это дополнение к нормальному виду наследования, где экземпляры подкласса наследуют методы суперкласса.) Таким образом, конструктор каждого подкласса Array также имеет унаследованное свойство по имени Symbol.species. (Или же подкласс при желании может определить собственное свойство с таким именем.)

- Методы вроде `map()` и `slice()`, которые создают и возвращают новые массивы, слегка подкорректированы в ES6 и последующих версиях. Вместо того чтобы создавать обыкновенный объект `Array`, для создания нового массива они (в действительности) вызывают

  `new this.constructor[Symbol.species]()`

Теперь перейдем к интересной части.

Предположим, что `Array[Symbol.species]` было просто обычным свойством данных, определенным примерно так:

```
Array[Symbol.species] = Array;
```

Тогда конструкторы подклассов унаследовали бы конструктор `Array()` как свой "вид" и вызов `map()` на подклассе массива возвращал бы экземпляр суперкласса, а не экземпляр подкласса. Тем не менее, на самом деле ES6 ведет себя не так. Причина в том, что `Array[Symbol.species]` представляет собой свойство с методами доступа, допускающее только чтение, функция получения которого просто возвращает `this`. Конструкторы подклассов наследуют эту функцию получения, т.е. конструктор каждого подкласса по умолчанию является собственным "видом".

Однако временами такое стандартное поведение не будет тем, что нужно. Если желательно, чтобы методы, возвращающие массивы, возвращали нормальные объекты `Array`, тогда вам понадобится установить `EZArray[Symbol.species]` в `Array`. Но так как унаследованное свойство предназначено только для чтения, его не удастся установить с помощью операции присваивания. Тем не менее, можно применить `defineProperty()`:

```
EZArray[Symbol.species] = Array; // Попытка установки свойства
                                 // только для чтения терпит неудачу.

// Взамен мы можем использовать defineProperty():
Object.defineProperty(EZArray, Symbol.species, {value: Array});
```

Самым простым вариантом, вероятно, будет явное определение собственного метода получения `Symbol.species` при создании класса:

```
class EZArray extends Array {
   static get [Symbol.species]() { return Array; }
   get first() { return this[0]; }
   get last() { return this[this.length-1]; }
}

let e = new EZArray(1,2,3);
let f = e.map(x => x - 1);
e.last  // => 3
f.last  // => undefined: f - нормальный массив без метода доступа last
```

Создание полезных подклассов `Array` было главным сценарием использования, который мотивировал введение `Symbol.species`, но это не единственное место, где применяется данный хорошо известный объект `Symbol`. Классы типизированных массивов используют его таким же способом, как и

класс Array. Аналогично метод slice() класса ArrayBuffer ищет свойство Symbol.species в this.constructor вместо того, чтобы просто создавать новый экземпляр ArrayBuffer. И методы Promise вроде then(), которые возвращают новые объекты Promise, создают их тоже через протокол "видов". Наконец, если вы сами создаете подкласс Map (например) и определяете методы, возвращающие новые объекты Map, то можете захотеть применить Symbol.species в интересах подклассов вашего подкласса.

14.4.5. Symbol.isConcatSpreadable

Метод concat() класса Array является одним из методов, описанных в предыдущем подразделе, которые используют Symbol.species для определения, какой конструктор задействовать для возвращаемого массива. Но в concat() также применяется Symbol.isConcatSpreadable. Вспомните из подраздела 7.8.3, что метод concat() массива обращается со своим значением this и аргументами типа массивов иначе, чем с аргументами не типа массивов. Аргументы, не являющиеся массивами, просто присоединяются к новому массиву, но массив this и любые аргументы типа массивов выравниваются или "распространяются", поэтому присоединяются элементы массива, а не сам массив.

До выхода ES6 метод concat() использовал Array.isArray() для выяснения, трактовать значение как массив или нет. В ES6 алгоритм слегка изменился: если аргумент (или значение this) метода concat() представляет собой объект и имеет свойство с символьным именем Symbol.isConcatSpreadable, тогда булевское значение этого свойства применяется для определения, должен ли аргумент "распространяться". Если такого свойства нет, то используется Array.isArray(), как было в предшествующих версиях языка.

Есть два случая, когда у вас может возникнуть желание задействовать Symbol.isConcatSpreadable.

- Если вы создаете объект, похожий на массив (см. раздел 7.9), и хотите, чтобы при передаче методу concat() он вел себя как настоящий массив, тогда можете просто добавить к своему объекту символьное свойство:

```
let arraylike = {
    length: 1,
    0: 1,
    [Symbol.isConcatSpreadable]: true
};
[].concat(arraylike)      // => [1]: (результат был бы [[1]],
                          //          если не распространять)
```

- Подклассы Array по умолчанию распространяемы, поэтому если вы определяете подкласс Array и не хотите, чтобы он вел себя как массив при передаче методу concat(), тогда можете[1] добавить в свой подкласс метод получения, подобный показанному ниже:

[1] Наличие ошибки в исполняющей среде JavaScript V8 не позволит этому коду корректно работать в Node 13.

```
class NonSpreadableArray extends Array {
   get [Symbol.isConcatSpreadable]() { return false; }
}
let a = new NonSpreadableArray(1,2,3);
[].concat(a).length       // => 1; (в случае распространения длина
                          //       составляла бы 3 элемента)
```

14.4.6. Объекты Symbol для сопоставления с образцом

В подразделе 11.3.2 были документированы строковые методы, которые выполняют операции сопоставления с образцом, используя аргумент RegExp. В ES6 и последующих версиях такие методы были обобщены, чтобы работать с объектами RegExp или любыми объектами, которые определяют поведение сопоставления с образцом через свойства с символьными именами. Для каждого строкового метода `match()`, `matchAll()`, `search()`, `replace()` и `split()` имеется соответствующий хорошо известный объект Symbol: `Symbol.match`, `Symbol.search` и т.д.

Объекты RegExp предлагают универсальный и очень мощный способ описания текстовых шаблонов, но они могут быть сложными и не особенно хорошо подходить для гибкого сопоставления. С помощью обобщенных строковых методов вы можете определять собственные классы шаблонов, применяющие символьные методы для обеспечения специального сопоставления. Скажем, вы могли бы выполнять сравнение строк с использованием объекта `Intl.Collator` (см. подраздел 11.7.3), чтобы при сопоставлении игнорировать диакритические знаки. Либо же вы могли бы определить класс шаблона на основе алгоритма *Soundex* для сопоставления слов на базе их сходного звучания в английском языке или для свободного сопоставления строк на заданном расстоянии Левенштейна.

В общем, когда вы вызываете один из этих пяти строковых методов с объектом шаблона следующего вида:

```
строка.метод(шаблон, аргумент)
```

такой вызов превращается в вызов метода с символическим именем на вашем объекте шаблона:

```
шаблон[символ](строка, аргумент)
```

В качестве примера давайте рассмотрим класс сопоставления с образцом, в котором применяются простые групповые символа * и ?, знакомые вам из файловых систем. Такой стиль сопоставления с образцом восходит к самым ранним дням существования операционной системы Unix и шаблоны часто называют *шаблонами поиска* (glob):

```
class Glob {
   constructor(glob) {
      this.glob = glob;

      // Мы реализуем сопоставление с шаблонами поиска,
      // внутренне используя RegExp.
```

```
        // ? соответствует любому одному символу кроме /,
        // а *соответствует нулю или большему количеству таких символов.
        // Для каждого применяется захватывающая группа.
        let regexpText =
              glob.replace("?", "([^/])").replace("*", "([^/]*)");
        // Мы используем флаг u для сопоставления,
        // осведомленного о кодировке Unicode.
        // Шаблоны поиска предназначены для сопоставления с целыми
        // строками, поэтому мы применяем якорные элементы ^ и $
        // и не реализуем search() или matchAll(),
        // т.к. они бесполезны с шаблонами подобного рода.
        this.regexp = new RegExp(`^${regexpText}$`, "u");
    }
    toString() { return this.glob; }
    [Symbol.search](s) { return s.search(this.regexp); }
    [Symbol.match](s) { return s.match(this.regexp); }
    [Symbol.replace](s, replacement) {
        return s.replace(this.regexp, replacement);
    }
}
let pattern = new Glob("docs/*.txt");
"docs/js.txt".search(pattern)  // => 0: соответствует в символе 0
"docs/js.htm".search(pattern)  // => -1: не соответствует
let match = "docs/js.txt".match(pattern);
match[0]                       // => "docs/js.txt"
match[1]                       // => "js"
match.index                    // => 0
"docs/js.txt".replace(pattern, "web/$1.htm")    // => "web/js.htm"
```

14.4.7. Symbol.toPrimitive

В подразделе 3.9.3 объяснялось, что в JavaScript есть три немного отличающихся алгоритма для преобразования объектов в элементарные значения. Грубо говоря, для преобразований, где строковое значение ожидается или предпочтительнее, интерпретатор JavaScript вызывает сначала метод `toString()` объекта и обращается к методу `valueOf()`, если `toString()` не определен или не возвращает элементарное значение. Для преобразований, в которых предпочтительнее числовое значение, интерпретатор JavaScript пробует сначала метод `valueOf()` и обращается к методу `toString()`, если метод `valueOf()` не определен или не возвращает элементарное значение. Наконец, в случаях, когда предпочтения отсутствуют, интерпретатор JavaScript позволяет классу решать, как делать преобразование. Объекты `Date` выполняют преобразование, используя сначала `toString()`, а все остальные типы пробуют сначала `valueOf()`.

В ES6 хорошо известный объект `Symbol` под названием `Symbol.toPrimitive` позволяет переопределять такое стандартное поведение преобразования объектов в элементарные значения и предоставляет вам полный контроль над тем,

каким образом экземпляры ваших классов будут преобразовываться в элементарные значения. Для этого определите метод с таким символьным именем.

Метод обязан возвращать элементарное значение, которое как-то представляет объект. Определенный вами метод будет вызываться с единственным строковым аргументом, который сообщает вам, преобразование какого вида интерпретатор JavaScript пытается выполнить в отношении вашего объекта.

- Если аргументом является `"string"`, то это значит, что интерпретатор JavaScript делает преобразование в контексте, в котором ожидается или предпочтительна (но не обязательна) строка. Подобное происходит, например, когда вы интерполируете объект в шаблонный литерал.
- Если аргументом является `"number"`, то это значит, что интерпретатор JavaScript делает преобразование в контексте, где ожидается или предпочтительно (но не обязательно) числовое значение. Подобное происходит, когда вы используете объект с операцией < или > либо с арифметическими операциями вроде - и *.
- Если аргументом является `"default"`, то это значит, что интерпретатор JavaScript преобразует ваш объект в контексте, в котором может работать либо числовое, либо строковое значение. Подобное происходит, когда вы используете объект с операциями +, == и !=.

Многие классы могут игнорировать описанный выше аргумент и просто возвращать одно и то же элементарное значение во всех случаях. Если вы хотите, чтобы экземпляры вашего класса поддерживали сравнение и сортировку с помощью < и >, тогда у вас есть веская причина определить метод `[Symbol.toPrimitive]`.

14.4.8. Symbol.unscopables

Последний хорошо известный объект `Symbol`, который мы раскроем здесь, не особо ясен и был введен как обходной прием для решения проблем с совместимостью, вызванных устаревшим оператором `with`. Вспомните, что оператор `with` принимает объект и выполняет свое тело, как если бы оно находилось в области видимости, где свойства этого объекта были переменными. Когда в класс `Array` были добавлены новые методы, возникли проблемы с совместимостью, из-за которых перестала работать часть существующего кода. Результатом стало появление объекта `Symbol.unscopables`. В ES6 и последующих версиях оператор `with` был слегка модифицирован. При использовании с объектом `o` оператор `with` вычисляет `Object.keys(o[Symbol.unscopables]||{})` и игнорирует свойства, имена которых присутствуют в результирующем массиве, когда создает искусственную область видимости, где выполняется его тело. Такое решение применяется в ES6 для добавления новых методов к `Array.prototype` без нарушения работы существующего кода в веб-сети. Это означает, что вы можете отыскать список самых новых методов класса `Array` посредством следующего вычисления:

```
let newArrayMethods = Object.keys(Array.prototype[Symbol.unscopables]);
```

14.5. Теги шаблонов

Строки внутри обратных кавычек известны как "шаблонные литералы" и были раскрыты в подразделе 3.3.4. Когда за выражением, значением которого является функция, следует шаблонный литерал, выражение превращается в вызов функции, называемый "тегированным шаблонным литералом" (tagged template literal). Определение новой теговой функции для использования с тегированными шаблонными литералами можно считать метапрограммированием, поскольку тегированные шаблоны часто применяются для определения проблемно-ориентированных языков (domain-specific language — DSL), и определение новой теговой функции подобно добавлению нового синтаксиса к JavaScript. Тегированные шаблонные литералы были приняты многими интерфейсными пакетами JavaScript. Язык запросов GraphQL использует теговую функцию `gql``` для того, чтобы запросы можно было встраивать в код JavaScript. Библиотека Emotion применяет теговую функцию `css``, чтобы сделать возможным встраивание стилей CSS в код JavaScript. В настоящем разделе будет показано, как писать собственные теговые функции такого рода.

В теговых функциях не ничего необычного: они являются обыкновенными функциями JavaScript, и для их определения не требуется никакого специального синтаксиса. Когда за выражением функции следует шаблонный литерал, функция вызывается. Первый аргумент представляет собой массив строк и за ним могут быть указаны ноль или большее количество дополнительных аргументов, имеющие значения любого типа.

Количество аргументов зависит от количества значений, которые интерполируются внутри шаблонного литерала. Если шаблонный литерал — просто постоянная строка, не содержащая интерполяций, тогда теговая функция будет вызвана с массивом из одной строки и без дополнительных аргументов. Если шаблонный литерал включает одно интерполированное значение, тогда теговая функция вызывается с двумя аргументами. В первом аргументе передается массив из двух строк, а во втором — интерполированное значение. Строки в начальном массиве — это строка слева от интерполированного значения и строка справа от него, причем любая одна из них может быть пустой строкой. Если шаблонный литерал включает два интерполированных значения, тогда теговая функция вызывается с тремя аргументами: массив из трех строк и два интерполированных значения. Три строки (любая из которых или все могут быть пустыми) — это текст слева от первого значения, текст между двумя значениями и текст справа от второго значения. В общем случае, если шаблонный литерал имеет n интерполированных значений, тогда теговая функция будет вызвана с n+1 аргументов. В первом аргументе передается массив из n+1 строк, а в остальных — n интерполированных значений в порядке, в котором они следуют в шаблонном литерале.

Значением шаблонного литерала всегда является строка. Но значением тегированного шаблонного литерала будет любое значение, возвращаемое теговой функцией. Им может быть строка, но когда теговая функция используется для

реализации DSL, возвращаемым значением обычно оказывается нестроковая структура данных, которая является разобранным представлением строки.

В качестве примера теговой функции для шаблона, возвращающей строку, рассмотрим показанный ниже шаблон html``, который полезен, если вы хотите безопасно интерполировать значения внутрь строки HTML-разметки. Теговая функция выполняет отмену специальных символов HTML для каждого значения, прежде чем задействовать его при построении финальной строки:

```
function html(strings, ...values) {
   // Преобразовать каждое значение в строку
   // и отменить специальные символы HTML.
   let escaped = values.map(v => String(v)
                            .replace("&", "&amp;")
                            .replace("<", "&lt;")
                            .replace(">", "&gt;")
                            .replace('"', "&quot;")
                            .replace("'", "&#39;"));

   // Возвратить объединенные строки и отмененные значения.
   let result = strings[0];
   for(let i = 0; i < escaped.length; i++) {
      result += escaped[i] + strings[i+1];
   }
   return result;
}

let operator = "<";
html`<b>x ${operator} y</b>`    // => "<b>x &lt; y</b>"

let kind = "game", name = "D&D";
html`<div class="${kind}">${name}</div>`
                                // =>'<div class="game">D&amp;D</div>'
```

Чтобы привести пример теговой функции, которая возвращает не строку, а разобранное представление строки, давайте вернемся к классу шаблона Glob, определенному в подразделе 14.4.6. Так как конструктор Glob() принимает одиночный строковый аргумент, мы можем определить теговую функцию для создания новых объектов Glob:

```
function glob(strings, ...values) {
   // Собрать строки и значения в единственную строку.
   let s = strings[0];
   for(let i = 0; i < values.length; i++) {
      s += values[i] + strings[i+1];
   }
   // Возвратить разобранное представление этой строки.
   return new Glob(s);
}

let root = "/tmp";
let filePattern = glob`${root}/*.html`; // Альтернатива объектам RegExp
"/tmp/test.html".match(filePattern)[1] // => "test"
```

Одним из средств, мимоходом упомянутых в подразделе 3.3.4, была теговая функция `String.raw``, которая возвращает строку в ее "необработанной" форме, не интерпретируя любые управляющие последовательности с обратной косой чертой. Она реализована с применением особенности вызова теговых функций, которую мы еще не обсуждали. Мы видели, что когда теговая функция вызывается, в ее первом аргументе передается массив строк. Но этот массив также имеет свойство по имени `raw`, значением которого является еще один массив строк с тем же самым количеством элементов. Массив в аргументе содержит строки, в которых управляющие последовательности были интерпретированы как обычно, а массив `raw` включает строки, где управляющие последовательности не интерпретировались. Такая малоизвестная особенность важна, если вы хотите определить DSL с грамматикой, которая использует обратные косые черты. Скажем, если нужно, чтобы теговая функция `glob`` поддерживала сопоставление с образцом для путей в стиле Windows (в которых применяются обратные косые черты, а не прямые) и нежелательно заставлять пользователей дублировать каждую обратную черту, то функцию `glob()` можно переписать с целью использования `strings.raw[]` вместо `strings[]`. Недостаток, конечно же, в том, что мы больше не сможем применять в шаблонных литералах управляющие последовательности вроде `\u`.

14.6. API-интерфейс `Reflect`

Объект `Reflect` не является классом; подобно объекту `Math` его свойства просто определяют коллекцию связанных функций. Такие функции, добавленные в ES6, определяют API-интерфейс для выполнения "рефлексии" объектов и их свойств. Новой функциональности здесь немного: объект `Reflect` определяет удобный набор функций, все в единственном пространстве имен, которые имитируют поведение синтаксиса базового языка и дублируют возможности разнообразных существующих функций `Object`.

Хотя функции `Reflect` не предлагают никаких новых возможностей, они группируют их вместе в одном удобном API-интерфейсе. И, что важнее, набор функций `Reflect` имеет отображение один к одному с набором методов обработчиков `Proxy`, которые мы рассмотрим в разделе 14.7.

API-интерфейс `Reflect` состоит их следующих функций.

- `Reflect.apply(f, o, args)`. Эта функция вызывает функцию `f` как метод объекта `o` (или вызывает ее как функцию без значения `this`, если `o` равно `null`) и передает значения массива `args` в качестве аргументов. Эквивалентна `f.apply(o, args)`.
- `Reflect.construct(c, args, newTarget)`. Эта функция вызывает конструктор `c`, как если бы использовалось ключевое слово `new`, и передает элементы массива `args` в качестве аргументов. Если необязательный аргумент `newTarget` указан, то он используется как значение `new.target` внутри вызова конструктора. Если он не указан, тогда значением `new.target` будет `c`.

- `Reflect.defineProperty(o, name, descriptor)`. Эта функция определяет свойство в объекте `o`, применяя `name` (строку или символ) как имя свойства. Объект `descriptor` должен определять значение (либо метод получения и/или метод установки) и атрибуты свойства. Функция `Reflect.defineProperty()` очень похожа на `Object.defineProperty()`, но возвращает `true` в случае успеха и `false` при неудаче. (`Object.defineProperty()` возвращает `o` в случае успеха и генерирует `TypeError` при неудаче.)
- `Reflect.deleteProperty(o, name)`. Эта функция удаляет свойство с указанным строковым или символьным именем из объекта `o`, возвращая `true` в случае успеха (или если такое свойство не существует) и `false`, если свойство удалить не удалось. Вызов данной функции похож на код `delete o[name]`.
- `Reflect.get(o, name, receiver)`. Эта функция возвращает значение свойства с указанным именем (строкой или символом) объекта `o`. Если свойство имеет метод получения и указан необязательный аргумент `receiver`, тогда функция получения вызывается как метод `receiver`, а не как метод `o`. Вызов данной функции подобен вычислению `o[name]`.
- `Reflect.getOwnPropertyDescriptor(o, name)`. Эта функция возвращает объект дескриптора свойства, который описывает атрибуты свойства по имени `name` объекта `o`, или `undefined`, если такое свойство не существует. Данная функция почти идентична `Object.getOwnPropertyDescriptor()` за исключением того, что версия функции из API-интерфейса `Reflect` требует, чтобы первый аргумент был объектом, и генерирует `TypeError`, если это не так.
- `Reflect.getPrototypeOf(o)`. Эта функция возвращает прототип объекта `o` или `null`, если объект не имеет прототипа. Она генерирует `TypeError`, если `o` является элементарным значением, а не объектом. Данная функция почти идентична `Object.getPrototypeOf()` за исключением того, что `Object.getPrototypeOf()` генерирует `TypeError` только для аргументов `null` и `undefined` и принудительно помещает остальные элементарные значения в их объекты-оболочки.
- `Reflect.has(o, name)`. Эта функция возвращает `true`, если объект `o` имеет свойство с именем, указанным в `name` (которое должно быть строкой или символом). Ее вызов подобен вычислению `name in o`.
- `Reflect.isExtensible(o)`. Эта функция возвращает `true`, если объект `o` является расширяемым (см. раздел 14.2), и `false`, если нет. Она генерирует `TypeError`, если `o` — не объект. `Object.isExtensible()` похожа, но просто возвращает `false`, когда в ее аргументе передается не объект.
- `Reflect.ownKeys(o)`. Эта функция возвращает массив имен свойств объекта `o` или генерирует `TypeError`, если `o` — не объект. Имена в возвращенном массиве будут строками и/или символами. Вызов данной функции подобен вызовам `Object.getOwnPropertyNames()` и `Object.getOwnPropertySymbols()` и объединяет их результаты.

- `Reflect.preventExtensions(o)`. Эта функция устанавливает атрибут `extensible` (см. раздел 14.2) объекта `o` в `false` и возвращает `true`, чтобы указать на успех. Она генерирует `TypeError`, если `o` — не объект. Функция `Object.preventExtensions()` дает тот же результат, но возвращает `o` вместо `true` и не генерирует `TypeError` для аргументов, не являющихся объектами.
- `Reflect.set(o, name, value, receiver)`. Эта функция устанавливает свойство объекта `o` с указанным в `name` именем в значение, указанное в `value`. Она возвращает `true` в случае успеха и `false` в случае неудачи (которая может произойти, если свойство допускает только чтение). Она генерирует `TypeError`, если `o` — не объект. Если указанное свойство является свойством с функцией установки и передан необязательный аргумент `receiver`, тогда функция установки будет вызвана как метод `receiver`, а не метод `o`. Вызов данной функции обычно представляет собой то же самое, что и вычисление `o[name] = value`.
- `Reflect.setPrototypeOf(o, p)`. Эта функция устанавливает прототип объекта `o` в `p`, возвращая `true` в случае успеха и `false` в случае неудачи (которая может произойти, если `o` не является расширяемым или операция породит циклическую цепочку прототипов). Она генерирует `TypeError`, если `o` — не объект или `p` — не объект и не `null`. Функция `Object.setPrototypeOf()` похожа, но возвращает `o` в случае успеха и генерирует `TypeError` в случае неудачи. Вспомните, что вызов любой из указанных функций, вероятно, сделает ваш код медленнее, разрушая оптимизацию со стороны интерпретатора JavaScript.

14.7. Объекты `Proxy`

Класс посредника `Proxy`, доступный в ES6 и последующих версиях, является наиболее мощным средством метапрограммирования в JavaScript. Он позволяет писать код, который изменяет фундаментальное поведение объектов JavaScript. Описанный в разделе 14.6 API-интерфейс `Reflect` представляет собой набор функций, которые дают нам прямой доступ к набору фундаментальных операций над объектами JavaScript. Класс `Proxy` предлагает нам способ реализовать такие фундаментальные операции самостоятельно и создать объекты, которые ведут себя так, как не могут вести обыкновенные объекты.

Когда мы создаем объект `Proxy`, то указываем два других объекта — объект цели (`target`) и объект обработчиков (`handlers`):

```
let proxy = new Proxy(target, handlers);
```

Результирующий объект `Proxy` не обладает собственным состоянием или поведением. Всякий раз, когда вы выполняете над ним операцию (читаете свойство, записываете свойство, определяете новое свойство, ищете прототип, вызываете его как функцию), он отправляет такие операции объекту обработчиков или объекту цели.

Посредники поддерживают те же самые операции, которые определены в API-интерфейсе Reflect. Пусть p — объект Proxy и вы написали delete p.x. Функция Reflect.deleteProperty() имеет такое же поведение, как операция delete. И когда вы используете операцию delete для удаления свойства объекта Proxy, он ищет метод deleteProperty() в объекте обработчиков. Если такой метод существует, то он вызывает его, а если нет, тогда объект Proxy выполняет удаление свойства в объекте цели.

Объекты Proxy работают описанным образом для всех фундаментальных операций: если соответствующий метод существует в объекте обработчиков, то он вызывается для выполнения операции. (Имена и сигнатуры методов совпадают с именами и сигнатурами функций Reflect, раскрытых в разделе 14.6.) И если такой метод не существует в объекте обработчиков, тогда объект Proxy выполняет фундаментальную операцию над объектом цели. Это означает, что объект Proxy способен получать свое поведение от объекта цели или от объекта обработчиков. Если объект обработчиков пуст, тогда посредник по существу является прозрачной оболочкой вокруг объекта цели:

```
let t = { x: 1, y: 2 };
let p = new Proxy(t, {});
p.x           // => 1
delete p.y    // => true: удаляет свойство y посредника
t.y           // => undefined: удаляет его также в объекте цели
p.z = 3;      // Определение нового свойства в посреднике
t.z           // => 3: определяет свойство в объекте цели
```

Прозрачный посредник-оболочка такого вида, по сути, эквивалентен внутреннему объекту цели, т.е. фактически нет причины его применять вместо помещенного в оболочку объекта. Однако прозрачные оболочки могут быть полезны, когда создаются как "аннулируемые посредники". Вместо создания объекта Proxy с помощью конструктора Proxy() вы можете применить фабричную функцию Proxy.revocable(), возвращающую объект, который включает объект Proxy и также функцию revoke(). Как только вы вызовите функцию revoke(), посредник немедленно прекращает работу:

```
function accessTheDatabase() { /* реализация опущена */ return 42; }
let {proxy, revoke} = Proxy.revocable(accessTheDatabase, {});

proxy()   // => 42: Посредник предоставляет доступ
          //        к внутренней функции цели.
revoke(); // Но при желании этот доступ можно отключить в любое время.
proxy();  // !TypeError: мы больше не можем вызывать эту функцию.
```

Следует отметить, что в дополнение к демонстрации аннулируемых посредников предыдущий код также демонстрирует, что посредники могут работать с функциями цели, равно как и с объектами цели. Но главный момент здесь в том, что аннулируемые посредники являются строительными блоками для своего рода изоляции кода, и вы можете использовать их, например, когда будете иметь дело с не заслуживающими доверия сторонними библиотеками. Если вам нужно передать свою функцию библиотеке, которую вы не контролируете, тогда взамен можете передать аннулируемого посредника и аннулировать его

по окончании работы с библиотекой. Тем самым вы предотвратите сохранение библиотекой ссылки на вашу функцию и вызов ее в непредсказуемое время. Защитное программирование такого рода не типично в программах JavaScript, но класс Proxy, по крайней мере, делает его возможным.

Если мы передаем конструктору `Proxy()` непустой объект обработчиков, то вместо определения прозрачного объекта-оболочки мы реализуем специальное поведение для посредника. При правильном наборе обработчиков внутренний объект цели по существу становится неважным.

Скажем, в следующем коде показано, как мы могли бы реализовать объект, который выглядит имеющим бесконечное количество свойств только для чтения, где значение каждого свойства совпадает с его именем:

```
// Мы используем объект Proxy для создания объекта,
// который выглядит имеющим любое возможное свойство,
// причем значение каждого свойства равно его имени.
let identity = new Proxy({}, {
    // Каждое свойство имеет собственное имя и значение.
    get(o, name, target) { return name; },
    // Каждое имя свойства определено.
    has(o, name) { return true; },
    // Слишком много свойств для перечисления,
    // поэтому мы просто генерируем исключение.
    ownKeys(o) {throw new RangeError("Infinite number of properties");},
                                        // Бесконечное количество свойств
    // Все свойства существуют и не являются записываемыми,
    // конфигурируемыми или перечислимыми.
    getOwnPropertyDescriptor(o, name) {
        return {
            value: name,
            enumerable: false,
            writable: false,
            configurable: false
        };
    },
    // Все свойства допускают только чтение, поэтому не могут быть установлены
    set(o, name, value, target) { return false; },
    // Все свойства не являются конфигурируемыми, поэтому их нельзя удалять
    deleteProperty(o, name) { return false; },
    // Все свойства существуют и не являются конфигурируемыми,
    // поэтому нельзя определять дополнительные свойства.
    defineProperty(o, name, desc) { return false; },
    // В сущности, это означает, что объект не является расширяемым.
    isExtensible(o) { return false; },
    // Все свойства уже определены в данном объекте, поэтому он не может
    // ничего наследовать, даже если бы у него был объект-прототип.
    getPrototypeOf(o) { return null; },
    // Объект не является расширяемым, поэтому мы не можем изменить
    // его прототип.
    setPrototypeOf(o, proto) { return false; },
});
```

```
identity.x              // => "x"
identity.toString       // => "toString"
identity[0]             // => "0"
identity.x = 1;         // Установка свойств безуспешна
identity.x              // => "x"
delete identity.x       // => false: также нельзя удалять свойства
identity.x              // => "x"
Object.keys(identity); //!RangeError: невозможно перечислить все ключи
for(let p of identity) ;//!RangeError
```

Объекты Proxy могут получать свое поведение от объекта цели и от объекта обработчиков, но в приведенных до сих пор примерах применялся один объект или другой. Тем не менее, обычно полезнее определять посредники, которые используют оба объекта.

Например, в показанном ниже коде с применением Proxy создается допускающая только чтение оболочка для объекта цели. Когда код пытается читать значения из объекта, операции чтения направляются объекту цели. Но если код пытается модифицировать объект или его свойства, тогда методы объекта обработчиков генерируют TypeError. Посредник такого рода может быть полезен при написании тестов: скажем, вы реализовали функцию, которая принимает в своем аргументе объект, и хотите удостовериться в том, что она не пытается модифицировать входной аргумент. Если ваш тест проходит в объекте-оболочке, допускающем только чтение, тогда любые операции записи будут приводить к генерации исключений, из-за которых тест потерпит неудачу:

```
function readOnlyProxy(o) {
   function readonly() { throw new TypeError("Readonly"); }
   return new Proxy(o, {
      set: readonly,
      defineProperty: readonly,
      deleteProperty: readonly,
      setPrototypeOf: readonly,
   });
}

let o = { x: 1, y: 2 };       // Нормальный записываемый объект
let p = readOnlyProxy(o);     // Его версия только для чтения
p.x                           // => 1: чтение свойств работает
p.x = 2;                      // !TypeError: изменять свойства нельзя
delete p.y;                   // !TypeError: удалять свойства нельзя
p.z = 3;                      // !TypeError: добавлять свойства нельзя
p.__proto__ = {};             // !TypeError: изменять прототип нельзя
```

Еще одна методика, используемая при написании посредников, предусматривает определение методов обработчиков, которые перехватывают операции над объектом, но по-прежнему делегируют их выполнение объекту цели. Функции API-интерфейса Reflect (см. раздел 14.6) имеют в точности те же сигнатуры, что и методы обработчиков, поэтому они облегчают такое делегирование.

Например, вот посредник, который делегирует выполнение всех операций объекту цели, но применяет методы обработчиков для регистрации операций:

```
/*
 * Возвращает объект Proxy, который является оболочкой для o,
 * делегирующей все операции объекту o после регистрации каждой операции.
 * objname - строка, которая будет присутствовать в регистрационных
 * сообщениях для идентификации объекта. Если o имеет собственные
 * свойства, значения которых являются объектами или функциями, то при
 * запросе значений таких свойств вы получите объект loggingProxy,
 * так что поведение регистрации этого посредника "заразно".
 */
function loggingProxy(o, objname) {
    // Определить обработчики для нашего регистрирующего объекта Proxy.
    // Каждый обработчик регистрирует сообщение и затем делегирует
    // выполнение операции объекту цели.
    const handlers = {
        // Это обработчик особого случая, потому что для собственных
        // свойств, чьи значения являются объектами или функциями,
        // он возвращает объект-посредник, а не само значение.
        get(target, property, receiver) {
            // Зарегистрировать операцию получения.
            console.log(`Handler get(${objname},${property.toString()})`);

            // Использовать API-интерфейс Reflect
            // для получения значения свойства.
            let value = Reflect.get(target, property, receiver);

            // Если свойство является собственным свойством объекта
            // цели и значением оказывается объект или функция,
            // тогда возвратить объект Proxy для него.
            if (Reflect.ownKeys(target).includes(property) &&
                (typeof value === "object" || typeof value === "function")) {
                return loggingProxy(value, `${objname}.${property.toString()}`);
            }
            // Иначе возвратить неизмененное значение.
            return value;
        },
        // В следующих трех методах нет ничего особенного:
        // они регистрируют операции и делегируют их выполнение
        // объекту цели. Они представляют собой особый случай
        // просто потому, что мы можем избежать регистрации объекта
        // получателя, что способно породить бесконечную рекурсию.
        set(target, prop, value, receiver) {
            console.log(
                `Handler set(${objname},${prop.toString()},${value})`);
            return Reflect.set(target, prop, value, receiver);
        },
        apply(target, receiver, args) {
            console.log(`Handler ${objname}(${args})`);
            return Reflect.apply(target, receiver, args);
        },
        construct(target, args, receiver) {
            console.log(`Handler ${objname}(${args})`);
            return Reflect.construct(target, args, receiver);
        }
    };
```

```
    // Мы можем автоматически сгенерировать остальные обработчики.
    // Метапрограммирование ведет к победе!
    Reflect.ownKeys(Reflect).forEach(handlerName => {
        if (!(handlerName in handlers)) {
            handlers[handlerName] = function(target, ...args) {
                // Зарегистрировать операцию.
                console.log(`Handler ${handlerName}(${objname},${args})`);
                // Делегировать выполнение операции.
                return Reflect[handlerName](target, ...args);
            };
        }
    });
    // Возвратить объект посредника для объекта,
    // используя эти регистрирующие обработчики.
    return new Proxy(o, handlers);
}
```

Определенная ранее функция `loggingProxy()` создает посредников, которые регистрируют все способы их использования. Если вы пытаетесь разобраться в том, как недокументированная функция работает с передаваемыми ей объектами, то применение регистрирующего посредника может помочь.

Рассмотрим следующие примеры, которые дают подлинное представление об итерации по массиву:

```
// Определить массив данных и объект со свойством типа функции.
let data = [10,20];
let methods = { square: x => x*x };

// Создать регистрирующие посредники для массива и объекта.
let proxyData = loggingProxy(data, "data");
let proxyMethods = loggingProxy(methods, "methods");

// Предположим, что мы хотим понять, как работает метод Array.map().
data.map(methods.square)              // => [100, 400]

// Первым делом испытаем его с регистрирующим посредником для массива.
proxyData.map(methods.square)         // => [100, 400]
// Вот вывод:
// Handler get(data,map)
// Handler get(data,length)
// Handler get(data,constructor)
// Handler has(data,0)
// Handler get(data,0)
// Handler has(data,1)
// Handler get(data,1)

// Теперь испытаем его с регистрирующим посредником для объекта с методом
data.map(proxyMethods.square)         // => [100, 400]

// Вот вывод:
// Handler get(methods,square)
// Handler methods.square(10,0,10,20)
// Handler methods.square(20,1,10,20)
```

```
// В заключение используем регистрирующий посредник,
// чтобы узнать протокол итерации.
for(let x of proxyData) console.log("Datum", x);

// Вот вывод:
// Handler get(data,Symbol(Symbol.iterator))
// Handler get(data,length)
// Handler get(data,0)
// Datum 10
// Handler get(data,length)
// Handler get(data,1)
// Datum 20
// Handler get(data,length)
```

Из первой порции вывода мы узнаем, что метод `Array.map()` явно проверяет существование каждого элемента массива (приводя к вызову обработчика `has()`) перед фактическим чтением значения элемента (которое запускает обработчик `get()`). Предположительно это сделано для того, чтобы можно было отличать несуществующие элементы массива от элементов, которые существуют, но не определены.

Вторая порция вывода может напомнить нам о том, что функция, переданная методу `Array.map()`, вызывается с тремя аргументами: значение элемента, индекс элемента и сам массив. (С нашим выводом связана одна проблема: метод `Array.toString()` не помещает квадратные скобки в свой вывод и регистрируемые сообщения были бы яснее, если бы включали список аргументов `(10,0, [10,20])`.)

Третья порция вывода показывает нам, что цикл `for/of` работает путем поиска метода с символьным именем `[Symbol.iterator]`. Она также демонстрирует то, что реализация этого итераторного метода в классе `Array` аккуратно проверяет длину массива на каждой итерации и не выдвигает допущение о неизменности длины массива во время итерации.

14.7.1. Инварианты `Proxy`

Определенная ранее функция `readOnlyProxy()` создает объекты `Proxy`, которые фактически заморожены: любая попытка изменить значение или атрибут свойства либо удалить свойства будет приводить к генерации исключения. Но поскольку объект цели не заморожен, мы обнаружим, что можем запрашивать посредник с помощью `Reflect.isExtensible()` и `Reflect.getOwnPropertyDescriptor()`, а это будет говорить нам о том, что мы должны иметь возможность устанавливать, добавлять и удалять свойства. Таким образом, функция `readOnlyProxy()` создает объекты в несогласованном состоянии. Мы могли бы устранить проблему, добавив обработчики `isExtensible()` и `getOwnPropertyDescriptor()`, или же просто смириться с незначительной несогласованностью подобного рода.

Однако API-интерфейс обработчиков `Proxy` позволяет определять объекты с крупными несогласованностями, и в таком случае сам класс `Proxy` будет пре-

пятствовать созданию объектов Proxy, которые являются критически несогласованными. В начале текущего раздела посредники были описаны как объекты без собственного поведения, потому что они просто направляют все операции объекту обработчиков и объекту цели. Но это не сосем так: после направления операции класс Proxy выполняет некоторый контроль корректности результата, чтобы удостовериться в том, что важные инварианты JavaScript не были нарушены. Если обнаруживается нарушение, тогда посредник сгенерирует исключение TypeError вместо того, чтобы позволить операции продолжаться.

Например, посредник для нерасширяемого объекта будет генерировать TypeError, если обработчик isExtensible() когда-либо возвратит true:

```
let target = Object.preventExtensions({});
let proxy = new Proxy(target, { isExtensible() { return true; }});
Reflect.isExtensible(proxy);  // !TypeError: нарушение инварианта
```

Соответственно объекты посредников для нерасширяемых объектов целей не могут иметь обработчик getPrototypeOf(), который возвращает что угодно кроме реального объекта-прототипа цели. Вдобавок если объект цели имеет незаписываемые и неконфигурируемые свойства, тогда класс Proxy будет генерировать TypeError, когда обработчик get() возвратит что-нибудь отличающееся от фактического значения:

```
let target = Object.freeze({x: 1});
let proxy = new Proxy(target, { get() { return 99; }});
proxy.x; //!TypeError: значение, возвращенное get(), не совпадает с целью
```

Класс Proxy навязывает несколько дополнительных инвариантов, подавляющее большинство которых связаны с нерасширяемыми объектами целей и неконфигурируемыми свойствами объекта цели.

14.8. Резюме

Ниже перечислены основные моменты, которые рассматривались в главе.

- Объекты JavaScript имеют атрибут extensible, а свойства объектов — атрибуты writable, enumerable и configurable плюс атрибуты value и get и/или set. Вы можете использовать эти атрибуты, чтобы "запирать" свои объекты разнообразными способами, включая создание "запечатанных" и "замороженных" объектов.
- В JavaScript определены функции, которые позволяют обходить цепочку прототипов объекта и даже изменять прототип объекта (хотя это может сделать ваш код медленнее).
- Свойства объекта Symbol имеют значения, являющиеся "хорошо известными символами", которые можно применять для имен свойств или методов в определяемых вами объектах и классах. Это дает возможность управлять тем, как ваш объект взаимодействует с языковыми средствами JavaScript и с базовой библиотекой. Например, хорошо известные объ-

екты `Symbol` позволяют делать ваши классы итерируемыми и контролировать строку, которая отображается при передаче экземпляра методу `Object.prototype.toString()`. До версии ES6 такая настройка была доступна только для собственных классов, встроенных в реализацию.

- Тегированные шаблонные литералы представляют собой синтаксис вызова функций, а определение новой теговой функции подобно добавлению нового синтаксиса литералов к языку. Определение теговой функции, которая производит разбор своего аргумента с шаблонной строкой, позволяет встраивать языки DSL внутрь кода JavaScript. Теговые функции также предоставляют доступ к необработанной форме строковых литералов, в которой обратные косые черты не имеют специального значения.
- Класс `Proxy` и связанный API-интерфейс `Reflect` делают возможным низкоуровневый контроль над фундаментальными линиями поведения объектов JavaScript. Объекты `Proxy` могут использоваться как необязательно аннулируемые оболочки для улучшения инкапсуляции кода и также могут применяться для реализации нестандартных линий поведения объектов (подобно ряду API-интерфейсов особого случая, определенных ранними версиями веб-браузеров).

ГЛАВА 15

JavaScript в веб-браузерах

Язык JavaScript был создан в 1994 году со специальной целью — сделать возможным динамическое поведение в документах, отображаемых веб-браузерами. С тех пор язык значительно эволюционировал и одновременно произошел бурный рост масштабов и возможностей веб-платформы. В наши дни программисты на JavaScript могут считать веб-сеть полнофункциональной платформой для разработки приложений. Веб-браузеры специализируются на отображении сформатированного текста и изображений, но подобно собственным операционным системам браузеры также предлагают другие службы, включая графику, воспроизведение видео- и аудиоклипов, взаимодействие с сетью, хранение и многопоточную обработку. JavaScript — это язык, который позволяет веб-приложениям использовать службы, предоставляемые веб-платформой, и в настоящей главе будет показано, как задействовать наиболее важные из них.

Глава начинается с описания программной модели веб-платформы, где объясняется, каким образом сценарии встраиваются внутрь HTML-страниц (см. раздел 15.1), и как код JavaScript запускается асинхронно событиями (см. раздел 15.2). В разделах, следующих после такого вводного материала, документируются базовые API-интерфейсы JavaScript, которые позволяют вашим веб-приложениям:

- управлять содержимым документа (см. раздел 15.3) и стилями (см. раздел 15.4);
- определять экранные позиции элементов документа (см. раздел 15.5);
- создавать многократно используемые компоненты пользовательского интерфейса (см. раздел 15.6);
- рисовать графические элементы (см. разделы 15.7 и 15.8);
- воспроизводить и генерировать звуки (см. раздел 15.9);
- организовывать навигацию и хронологию браузера (см. раздел 15.10);
- обмениваться данными через сеть (см. раздел 15.11);
- сохранять данные на компьютере пользователя (см. раздел 15.12);
- выполнять параллельные вычисления с помощью потоков (см. раздел 15.13).

JavaScript стороны клиента

В этой книге и в веб-сети вы будете встречать термин "JavaScript стороны клиента". Он является просто синонимом кода JavaScript, написанного для запуска в веб-браузере, и применяется как противоположность коду "стороны сервера", который выполняется на веб-серверах.

Две "стороны" относятся к двум концам сетевого подключения, которые разделяют веб-сервер и веб-браузер, а разработка программного обеспечения для веб-сети обычно требует написания кода для обеих "сторон". Сторону клиента и сторону сервера также часто называют "интерфейсной частью" и "серверной частью".

В предшествующих изданиях книги предпринимались попытки всесторонне охватить все API-интерфейсы JavaScript, определяемые веб-браузерами, в результате чего десять лет тому назад эта книга была слишком объемной. Количество и сложность API-интерфейсов для веб-сети продолжили расти, и я больше не думаю, что имеет смысл пытаться раскрыть их все в одной книге. Что касается 7-го издания, то моя цель — полностью охватить язык JavaScript и предоставить исчерпывающее введение в использование языка со средой Node и веб-браузерами. В данной главе невозможно раскрыть все API-интерфейсы для веб-сети, но наиболее важные из них будут представлены достаточно подробно, чтобы вы могли немедленно приступить к работе с ними. Вдобавок, изучив описанные здесь базовые API-интерфейсы, вы должны быть в состоянии освоить новые API-интерфейсы (вроде упоминаемых в разделе 15.15), когда они вам понадобятся.

Среда Node имеет единственную реализацию и единственный авторитетный источник документации. Напротив, API-интерфейсы для веб-сети определены по соглашению между крупными поставщиками веб-браузеров, и авторитетная документация имеет форму спецификации, предназначенной для программистов на C++, которые реализуют API-интерфейсы, а не для программистов на JavaScript, их использующих. К счастью, проект "MDN web docs" корпорации Mozilla (`https://developer.mozilla.org/`) является надежным и исчерпывающим источником[1] документации по API-интерфейсам для веб-сети.

[1] Предшествующие издания книги содержали обширный справочный раздел, посвященный стандартной библиотеке JavaScript и API-интерфейсам для веб-сети. В 7-е издание он не был включен, поскольку проект "MDN web docs" сделал его непрактичным: теперь быстрее найти что-то в MDN, нежели листать страницы книги, а мои бывшие коллеги в MDN лучше справляются с поддержанием онлайновой документации в актуальном состоянии, чем когда-либо могла эта книга.

В течение 25 лет, прошедших с первого выпуска JavaScript, поставщики браузеров добавляли функциональные средства и API-интерфейсы для потребления программистами. Многие из этих API-интерфейсов уже устарели, включая перечисленные ниже.

- Патентованные API-интерфейсы, которые никогда не были стандартизированы и/или реализованы другими поставщиками браузеров. Много таких API-интерфейсов определено в Microsoft Internet Explorer. Некоторые (наподобие свойства `innerHTML`) оказались полезными и в итоге были стандартизированы. Остальные (вроде метода `attachEvent()`) с годами устарели.
- Неэффективные API-интерфейсы (например, метод `document.write()`), которые оказывают настолько серьезное влияние на производительность, что их применение больше не считается приемлемым.
- Устаревшие API-интерфейсы, которые давно были заменены новыми API-интерфейсами, достигающими той же цели. Примером служит свойство `document.bgColor`, которое было определено, чтобы позволить коду JavaScript устанавливать цвет фона документа. С появлением CSS свойство `document.bgColor` стало старомодным особым случаем, не имеющим реальной цели.
- Неудачно спроектированные API-интерфейсы, которые были заменены лучшими вариантами. На заре веб-сети комитеты по стандартам определили ключевой API-интерфейс DOM (Document Object Model — объектная модель документа) независимым от языка способом, чтобы тот же самый API-интерфейс можно было использовать в программах Java для работы с XML-документами и в программах JavaScript для работы с HTML-документами. Результатом стал API-интерфейс, который не очень хорошо подходил для языка JavaScript и располагал возможностями, которые не особо интересовали программистов веб-приложений. Потребовались десятилетия, чтобы избавиться от таких ранних проектных просчетов, но современные веб-браузеры поддерживают значительно более совершенную модель DOM.

Поставщики браузеров могут нуждаться в поддержке таких унаследованных API-интерфейсов в обозримом будущем, чтобы гарантировать обратную совместимость, но больше нет необходимости обсуждать их в этой книге или изучать самостоятельно. Веб-платформа обрела зрелость и стабилизировалась, а если вы — опытный разработчик веб-приложений и помните 4-е или 5-е издание этой книги, то обладаете многими устаревшими сведениями, о которых стоит забыть, т.к. есть новый материал для изучения.

15.1. Основы программирования для веб-сети

В текущем разделе объясняется, как структурированы программы JavaScript для веб-сети, каким образом они загружаются в веб-браузер, как они производят вывод и каким образом запускаются асинхронно, реагируя на события.

15.1.1. Код JavaScript в HTML-дескрипторах <script>

Веб-браузеры отображают HTML-документы. Если вы хотите, чтобы веб-браузер выполнил код JavaScript, тогда должны включить желаемый код в HTML-документ (или сослаться на код), и именно это делает HTML-дескриптор <script>.

Код JavaScript может быть встроен внутрь HTML-файла между дескрипторами <script> и </script>. Например, ниже показано содержимое HTML-файла, включающего дескриптор <script> с кодом JavaScript, который динамически обновляет один элемент документа, чтобы он вел себя как цифровые часы:

```
<!DOCTYPE html>              <!-- Это файл HTML5. -->
<html>                       <!-- Корневой элемент. -->
<head>                       <!-- Здесь могут находиться заголовок,
                                  сценарии и стили. -->
<title>Digital Clock</title>
<style>                      /* Таблица стилей CSS для часов. */
#clock {                     /*Стили применяются к элементу с id="clock"*/
  font: bold 24px sans-serif; /* Использовать крупный шрифт
                                 с полужирным начертанием */
  background: #ddf;          /* на светлом голубовато-сером фоне. */
  padding: 15px;             /* Окружить часы некоторым пространством */
  border: solid black 2px;   /* и сплошной черной рамкой */
  border-radius: 10px;       /* со скругленными углами. */
}
</style>
</head>
<body>              <!-- Тело вмещает в себе содержимое документа. -->
<h1>Digital Clock</h1>  <!-- Отобразить заголовок. -->
<span id="clock"></span> <!-- В этот элемент мы будем вставлять время -->
<script>
// Определить функцию для отображения текущего времени.
function displayTime() {
    let clock = document.querySelector("#clock"); // Получить элемент
                                                  // с id="clock".
    let now = new Date();                       // Получить текущее время.
    clock.textContent = now.toLocaleTimeString(); // Отобразить время
                                                  // в часах.
}
displayTime()                    // Немедленно отобразить время
setInterval(displayTime, 1000);// и затем обновлять его каждую секунду.
</script>
</body>
</html>
```

Хотя код JavaScript можно встраивать прямо внутрь дескриптора <script>, чаще с применением атрибута src дескриптора <script> указывается URL (абсолютный URL или URL относительно URL, где находится отображаемый HTML-файл) файла, содержащего код JavaScript. Если мы изымем код JavaScript из HTML-файла и сохраним его в собственный файл scripts/digital_clock.js, то вот как дескриптор <script> может ссылаться на файл кода:

```
<script src="scripts/digital_clock.js"></script>
```

Файл JavaScript содержит чистый код JavaScript без дескрипторов <script> или любой другой HTML-разметки. По соглашению файлы кода JavaScript имеют имена, которые заканчиваются на .js.

Дескриптор <script> с атрибутом src ведет себя в точности так, как если бы содержимое указанного файла JavaScript располагалось непосредственно между дескрипторами <script> и </script>. Обратите внимание, что закрывающий дескриптор </script> обязателен в HTML-документах, даже когда присутствует атрибут src: в HTML не поддерживается дескриптор <script/>.

С использованием атрибута src связано несколько преимуществ.

- Он упрощает HTML-файлы, позволяя убрать из них крупные блоки кода JavaScript, т.е. помогает отделять содержимое от поведения.
- Когда тот же самый код JavaScript применяется во множестве веб-страниц, то использование атрибута src дает возможность поддерживать только одну копию этого кода, а не редактировать каждый HTML-файл в случае изменения кода.
- Если файл кода JavaScript применяется более чем одной страницей, тогда он должен быть загружен только раз первой страницей, которая его использует — последующие страницы могут извлекать его из кеша браузера.
- Поскольку атрибут src принимает в качестве своего значения произвольный URL, программа JavaScript или веб-страница из одного веб-сервера может потреблять код, экспортированный другими веб-серверами. Большое количество рекламы в Интернете опирается на данный факт.

Модули

В разделе 10.3 были описаны модули JavaScript и их директивы import и export. Если вы написали свою программу JavaScript с применением модулей (и не использовали инструмент пакетирования кода для объединения всех модулей в единственный немодульный файл кода JavaScript), тогда должны загружать модуль верхнего уровня программы с помощью дескриптора <script>, который имеет атрибут type="module". В таком случае будет загружаться указанный вами модуль, все модули, которые он импортирует, и (рекурсивно) все импортируемые ими модули. За дополнительными деталями обращайтесь в раздел 10.3.5.

Указание типа сценария

На заре существования веб-сети считалось, что браузеры когда-нибудь смогут реализовывать языки, отличающиеся от JavaScript, и программисты добавляли к своим дескрипторам <script> атрибуты вроде language="javascript" и type="application/javascript". Это совершенно излишне. JavaScript — стандартный (и единственный) язык веб-сети. Атрибут language устарел, а для применения атрибута type в дескрипторе <script> есть лишь две причины:

- чтобы указать, что сценарий является модулем;
- чтобы внедрить данные внутрь веб-страницы, не отображая их (см. подраздел 15.3.4).

Запуск сценария: async и defer

Когда язык JavaScript впервые добавлялся в веб-браузеры, не существовало API-интерфейса для обхода и манипулирования структурой и содержимым уже визуализированного документа. Единственным способом воздействия кода JavaScript на содержимое документа была генерация содержимого на лету, пока документ находился в процессе загрузки. Цель достигалась за счет использования метода document.write() для внедрения HTML-текста внутрь документа в месте нахождения сценария.

Применение метода document.write() больше не считается хорошим стилем. Однако сам факт такой возможности означает, что когда синтаксический анализатор HTML встречает элемент <script>, он обязан по умолчанию запустить сценарий, просто чтобы удостовериться в отсутствии вывода любой HTML-разметки, прежде чем можно будет возобновить анализ и визуализацию документа. В итоге анализ и визуализация веб-страницы могут значительно замедлиться.

К счастью, такой стандартный *синхронный* или *блокирующий* режим выполнения сценария — не единственный вариант. Дескриптор <script> может иметь атрибуты defer и async, которые заставляют сценарии выполняться по-разному. Они являются булевскими атрибутами, т.е. не имеют значения; им нужно лишь присутствовать в дескрипторе <script>. Обратите внимание, что упомянутые атрибуты будут осмысленными, только когда используются в сочетании с атрибутом src:

```
<script defer src="deferred.js"></script>
<script async src="async.js"></script>
```

Оба атрибута, defer и async, представляют собой способы сообщения браузеру о том, что в связанном сценарии не применяется метод document.write() для генерации вывода HTML, вследствие чего браузер может продолжить синтаксический анализ и визуализацию документа во время загрузки сценария. Атрибут defer заставляет браузер отложить выполнение сценария до тех пор, пока документ полностью не загрузится, будет проанализирован и станет готовым к манипуляциям. Атрибут async заставляет браузер запустить сценарий как можно раньше, но не блокировать анализ документа во время загрузки сценария. Когда дескриптор <script> содержит оба атрибута, приоритет имеет async.

Обратите внимание, что отложенные сценарии запускаются в порядке, в котором они появляются в документе. Асинхронные сценарии запускаются по мере загрузки, т.е. могут выполняться не по порядку.

Сценарии с атрибутом `type="module"` по умолчанию выполняются после загрузки документа, как если бы они имели атрибут `defer`. Вы можете переопределить такое поведение посредством атрибута `async`, который заставит код выполняться, как только модуль и все его зависимости будут загружены.

Простая альтернатива атрибутам `async` и `defer`, особенно для кода, включенного прямо в HTML-разметку, предусматривает размещение сценариев в конце HTML-файла. Таким образом, сценарий может запускаться, зная о том, что содержимое документа перед ним было проанализировано и готово к манипуляциям.

Загрузка сценариев по запросу

Иногда у вас может быть код JavaScript, который не используется при первой загрузке документа и необходим лишь тогда, когда пользователь предпринимает какое-то действие вроде щелчка на кнопке или открытия меню. Если вы разрабатываете код с применением модулей, то можете загружать модуль по запросу с помощью `import()`, как было описано в подразделе 10.3.6.

Если вы не используете модули, тогда можете загружать файл кода JavaScript по запросу, просто добавляя в документ дескриптор `<script>`, когда сценарий нужно загрузить:

```
// Асинхронно загружает и выполняет сценарий из указанного URL.
// Возвращает объект Promise, который разрешается, когда сценарий загружен
function importScript(url) {
   return new Promise((resolve, reject) => {
      let s = document.createElement("script"); // Создать
                                                // элемент <script>.
      s.onload = () => { resolve(); };  // Разрешить объект Promise,
                                        // когда сценарий загружен.
      s.onerror = (e) => { reject(e); }; // Отклонить объект Promise
                                        // в случае неудачи.
      s.src = url;                      // Установить URL сценария.
      document.head.append(s);          //Добавить <script> в документ
   });
}
```

Функция `importScript()` применяет API-интерфейсы DOM (см. раздел 15.3) для создания нового дескриптора `<script>` и его добавления к дескриптору `<head>` документа. К тому же она использует обработчики событий (см. раздел 15.2) для установления, когда сценарий был загружен успешно или когда загрузка потерпела неудачу.

15.1.2. Объектная модель документа

Одним из наиболее важных объектов в программировании на JavaScript стороны клиента является объект `Document`, который представляет HTML-

документ, отображаемый в окне или на вкладке браузера. API-интерфейс для работы с HTML-документами известен как объектная модель документа (Document Object Model — DOM) и подробно обсуждается в разделе 15.3. Но модель DOM настолько важна при программировании на JavaScript стороны клиента, что заслуживает того, чтобы быть введенной прямо здесь.

HTML-документы содержат вложенные друг в друга HTML-элементы, формирующие дерево. Рассмотрим следующий простой HTML-документ:

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

Дескриптор <html> верхнего уровня содержит дескрипторы <head> и <body>. Дескриптор <head> содержит дескриптор <title>. Дескриптор <body> содержит дескрипторы <h1> и <p>. Дескрипторы <title> и <h1> содержат строки текста, а дескриптор <p> — две строки текста с дескриптором <i> между ними.

API-интерфейс DOM отражает древовидную структуру HTML-документа. Для каждого HTML-дескриптора в документе имеется соответствующий объект Element в JavaScript и для каждого фрагмента текста в документе есть соответствующий объект Text. Классы Element и Text и сам класс Document являются подклассами более универсального класса Node, а объекты Node организованы в древовидную структуру, которую код JavaScript может запрашивать с применением API-интерфейса DOM. Представление DOM примера документа выглядит как дерево, изображенное на рис. 15.1.

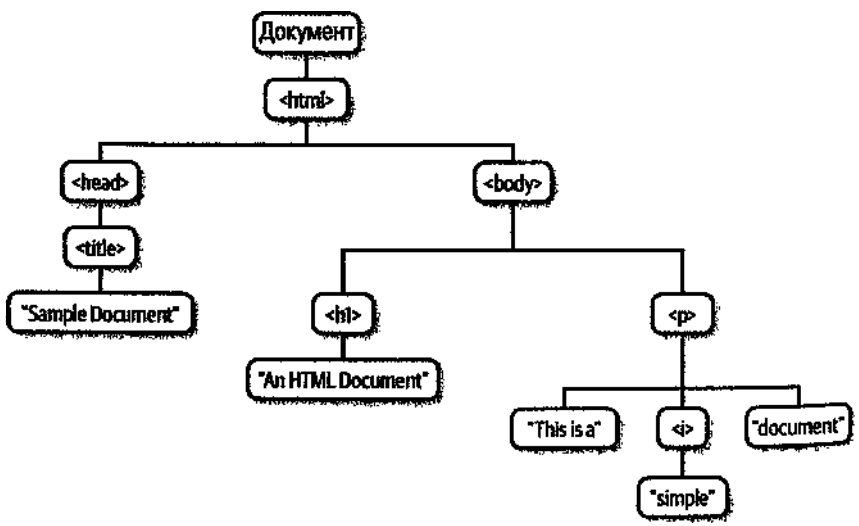


Рис. 15.1. Древовидное представление HTML-документа

Если вы еще не знакомы с древовидными структурами в компьютерном программировании, тогда полезно знать, что они заимствуют терминологию из родословных деревьев. Узел непосредственно выше какого-то узла является родителем этого узла. Узлы на уровень ниже еще одного узла являются дочерними для этого узла. Узлы на том же самом уровне и с тем же самым родителем являются родственными. Набор узлов, расположенных на любое количество уровней ниже отдельного узла, являются потомками данного узла. А родитель, прародитель и все остальные узлы выше некоторого узла являются его предками.

API-интерфейс DOM включает методы для создания новых узлов `Element` и `Text`, а также для их вставки в документ как дочерних по отношению к другим объектам `Element`. Вдобавок есть методы для перемещения элементов внутри документа и для их полного удаления. В то время как приложение стороны сервера может производить простой текстовый вывод, записывая строки с помощью `console.log()`, приложение JavaScript стороны клиента способно вырабатывать форматированный HTML-вывод за счет построения или манипулирования деревом документа с использованием API-интерфейса DOM.

Каждому типу HTML-дескриптора соответствует класс JavaScript и каждое вхождение дескриптора в документе представлено экземпляром этого класса. Например, дескриптор `<body>` представлен экземпляром `HTMLBodyElement`, а дескриптор `<table>` — экземпляром `HTMLTableElement`. Объекты элементов JavaScript имеют свойства, которые соответствуют атрибутам HTML-дескрипторов. Скажем, экземпляры `HTMLImageElement`, представляющие дескрипторы `<img>`, имеют свойство `src`, которое соответствует атрибуту `src` дескриптора. Начальным значением свойства `src` будет значение атрибута, которое указано в HTML-дескрипторе, и установка этого свойства в коде JavaScript изменяет значение атрибута HTML-дескриптора (плюс вынуждает браузер загрузить и отобразить новое изображение). Большинство классов элементов JavaScript просто отражают атрибуты HTML-дескрипторов, но в некоторых определены дополнительные методы. Например, в классах `HTMLAudioElement` и `HTMLVideoElement` определены методы наподобие `play()` и `pause()` для управления воспроизведением аудио- и видеоклипов.

15.1.3. Глобальный объект в веб-браузерах

Существует один глобальный объект для окна или вкладки браузера (см. раздел 3.7). Весь код JavaScript (кроме кода, запущенного в потоках веб-воркеров; см. раздел 15.13), выполняющийся в данном окне, разделяет единственный глобальный объект. Это верно вне зависимости от того, сколько сценариев или модулей имеется в документе: все сценарии и модули разделяют единственный глобальный объект; если один сценарий определяет свойство в глобальном объекте, то оно будет видимым также и всем остальным сценариям.

Именно в глобальном объекте определена стандартная библиотека JavaScript — функция `parseInt()`, объект `Math`, класс `Set` и т.д. В веб-браузерах глобальный объект также содержит главные точки входа различных API-интерфейсов для веб-сети. Скажем, свойство `document` представляет документ, отображаемый в

текущий момент, метод fetch() делает сетевые HTTP-запросы и конструктор Audio() позволяет программам JavaScript воспроизводить звуки.

В веб-браузерах глобальный объект выполняет двойную работу: в дополнение к определению встроенных типов и функций он также представляет текущее окно веб-браузера и определяет такие свойства, как history (см. подраздел 15.10.2), которое представляет хронологию просмотра окна, и innerWidth, хранящее ширину окна в пикселях. Одно из свойств глобального объекта называется window, а его значением является сам глобальный объект. Таким образом, для ссылки на глобальный объект в коде стороны клиента вы можете просто набрать window. При работе с функциями, специфичными для окна, часто бывает полезно включать префикс window.: например, window.innerWidth выглядит яснее, чем innerWidth.

15.1.4. Сценарии разделяют пространство имен

Благодаря модулям константы, переменные, функции и классы, определенные на верхнем уровне (т.е. за пределами любого определения функции или класса) модуля, являются закрытыми по отношению к модулю, если только они явно не экспортируются и тогда могут избирательно импортироваться другими модулями. (Следует отметить, что такая характеристика модулей принимается во внимание также инструментами пакетирования кода.)

Тем не менее, в немодульных сценариях ситуация совершенно иная. Если код верхнего уровня определяет константу, переменную, функцию или класс, то такое определение будет видимым для всех остальных сценариев в том же самом документе. Если в одном сценарии определяется функция f(), а в другом — класс C, тогда в третьем сценарии можно вызывать функцию f() и создавать экземпляр класса C, не предпринимая никаких действий по их импортированию. Таким образом, если вы не применяете модули, тогда независимые сценарии в вашем документе разделяют единственное пространство имен и ведут себя так, словно они все являются частью одного крупного сценария. Для небольших программ это может быть удобно, но избегание конфликтов имен нередко становится проблематичным в более крупных программах, особенно когда некоторые сценарии относятся к сторонним библиотекам.

Существует ряд исторически сложившихся особенностей работы такого разделяемого пространства имен. Объявления var и function на верхнем уровне создают свойства в разделяемом глобальном объекте. Если в одном сценарии на верхнем уровне определена функция f(), то другой сценарий в том же документе может вызывать ее как f() или как window.f(). С другой стороны, объявления const, let и class из ES6 на верхнем уровне не создают свойств в глобальном объекте. Однако они по-прежнему определены в разделяемом пространстве имен: если в одном сценарии определяется класс C, то другие сценарии будут иметь возможность создавать экземпляры класса C посредством new C(), но не new window.C().

Подводя итоги: в модулях объявления верхнего уровня имеют область видимости модуля и могут явно экспортироваться. Тем не менее, в немодульных

сценариях объявления верхнего уровня имеют область видимости вмещающего документа и разделяются всеми сценариями в документе. Более старые объявления `var` и `function` разделяются через свойства глобального объекта. Более новые объявления `const`, `let` и `class` также разделяются и имеют ту же область видимости документа, но не существуют в виде свойств какого-то объекта, к которому код JavaScript имел бы доступ.

15.1.5. Выполнение программ JavaScript

Формальное определение программы в JavaScript стороны клиента отсутствует, но мы можем сказать, что программа JavaScript состоит из всего кода JavaScript, который находится или на который производится ссылка в документе. Эти отдельные фрагменты кода разделяют единственный глобальный объект `window`, который дает им доступ к одному и тому же внутреннему объекту `Document`, представляющему HTML-документ. Немодульные сценарии дополнительно разделяют пространство имен верхнего уровня.

Если веб-страница включает встроенный фрейм (используя элемент `<iframe>`), то код JavaScript во встроенном документе имеет не такой глобальный объект и объект `Document`, как код в документе, содержащем встроенный документ, и может считаться отдельной программой JavaScript. Однако вспомните, что не существует формального определения границ программы JavaScript. Если контейнерный документ и содержащийся в нем документ загружаются с того же самого сервера, то код в одном документе может взаимодействовать с кодом в другом документе, и при желании вы можете трактовать их как две взаимодействующие части одной программы. В подразделе 15.13.6 будет показано, как программа JavaScript может отправлять и получать сообщения в и из кода JavaScript, выполняемого в `<iframe>`.

Вы можете считать, что выполнение программы JavaScript происходит в два этапа. На первом этапе загружается содержимое документа и запускается код из элементов `<script>` (как встроенные, так и внешние сценарии). Сценарии обычно запускаются в порядке, в котором они появляются в документе, хотя такой стандартный порядок можно изменять с помощью описанных ранее атрибутов `async` и `defer`. Код JavaScript внутри любого отдельного сценария выполняется сверху вниз, подчиняясь, конечно же, условным операторам, циклам и другим управляющим операторам JavaScript. Некоторые сценарии в действительности ничего не делают на первом этапе, а просто определяют функции и классы для применения на втором этапе. Другие сценарии могут выполнять значительный объем работ на первом этапе и ничего не делать на втором. Вообразим себе сценарий в самом конце документа, который ищет все дескрипторы `<h1>` и `<h2>` в документе и модифицирует документ, генерируя и вставляя оглавление в начало документа. Всю работу можно было бы сделать на первом этапе. (Пример приводится в подразделе 15.3.6.)

После того, как документ загружен, и все сценарии выполнены, выполнение программы JavaScript переходит ко второму этапу, который является асинхронным и управляемым событиями. Если сценарий собирается принимать участие во втором этапе, тогда одним из действий, которое он обязан предпринять на

первом этапе, должна быть регистрация, по крайней мере, одной функции обработчика событий или другой функции обратного вызова, которая будет вызываться асинхронно. На этом втором управляемом событиями этапе веб-браузер вызывает функции обработчиков событий и другие обратные вызовы в ответ на события, которые происходят асинхронно. Обработчики событий чаще всего вызываются в ответ на пользовательский ввод (щелчки кнопками мыши, нажатия клавиш и т.д.), но многие также запускаются активностью сети, загрузкой документов и ресурсов, истекшим временем или ошибками в коде JavaScript. События и обработчики событий подробно описаны в разделе 15.2.

В число первых событий, которые происходят на управляемом событиями этапе, входят события `DOMContentLoaded` и `load`. Событие `DOMContentLoaded` инициируется, когда HTML-документ был полностью загружен и синтаксически проанализирован. Событие `load` инициируется, когда все внешние ресурсы документа, такие как изображения, также полностью загружены. Программы JavaScript часто используют одно из указанных событий в качестве спускового механизма или стартового сигнала. Вполне обычно видеть программы, сценарии которых определяют функции, но не предпринимают никаких действий, кроме регистрации функции обработчика событий, подлежащего запуску событием `load` в начале управляемого событиями этапа выполнения. Именно этот обработчик событий `load` затем манипулирует документом и делает все то, что должна делать программа. Следует отметить, что в программировании на JavaScript функция обработчика событий наподобие описанного здесь обработчика событий `load` часто регистрирует другие обработчики событий.

Этап загрузки программы JavaScript относительно короткий: в идеале меньше секунды. После того, как документ загружен, управляемый событиям этап длится до тех пор, пока документ отображается веб-браузером. Поскольку этот этап асинхронный и управляемый событиями, могут случаться длительные периоды бездействия, когда никакой код JavaScript не выполняется, перемежающиеся всплесками активности, которые вызваны событиями от пользователя или сети. Мы рассмотрим такие два этапа более детально далее в главе.

Потоковая модель JavaScript стороны клиента

JavaScript — однопоточный язык, и однопоточное выполнение значительно упрощает программирование: вы можете писать код с уверенностью, что два обработчика событий никогда не запустятся одновременно. Вы можете манипулировать содержимым документа, зная о том, что никакой другой поток не попытается его модифицировать в то же самое время, и при написании кода JavaScript вам никогда не придется беспокоиться о блокировках, взаимоблокировках или состязаниях.

Однопоточное выполнение означает, что во время выполнения сценариев и обработчиков событий веб-браузеры перестают реагировать на пользовательский ввод. В итоге ответственность возлагается на программистов на JavaScript: это значит, что сценарии и обработчики событий JavaScript не должны выполняться слишком долго. Если сценарий выполняет задачу с интенсивными вычислениями, то он внесет задержку в загрузку документа и пользователь не

увидит содержимое документа до тех пор, пока сценарий не завершится. Если обработчик событий выполняет задачу с интенсивными вычислениями, то браузер может перестать реагировать, что возможно заставит пользователя думать о его аварийном отказе.

Веб-платформа определяет управляемую форму параллелизма, называемую "веб-воркером" (web worker). Веб-воркер — это фоновый поток для выполнения задач с интенсивными вычислениями без замораживания пользовательского интерфейса. Код, который выполняется в потоке веб-воркера, не имеет доступа к содержимому документа, не разделяет никакого состояния с главным потоком или другими веб-воркерами и может взаимодействовать с главным потоком и другими веб-воркерами только посредством событий асинхронных сообщений. Таким образом, параллелизм не поддается обнаружению главным потоком, а веб-воркеры не меняют базовую модель однопоточного выполнения программ JavaScript. Безопасный механизм потоков веб-воркеров обсуждается в разделе 15.13.

Временная шкала JavaScript стороны клиента

Вы уже видели, что программы JavaScript начинаются с этапа выполнения сценариев и затем переходят к этапу обработки событий. Указанные два этапа можно дополнительно разбить на следующие шаги.

1. Веб-браузер создает объект `Document` и начинает синтаксический анализ веб-страницы, добавляя в документ объекты `Element` и узлы `Text` по мере того, как он анализирует HTML-элементы и их текстовое содержимое. Свойство `document.readyState` имеет на этом этапе значение `loading` (загружается).

2. Когда синтаксический анализатор HTML встречает дескриптор `<script>`, который не имеет атрибутов `async`, `defer` или `type="module"`, он добавляет такой дескриптор сценария в документ и затем запускает сценарий. Сценарий выполняется синхронно, а синтаксический анализатор HTML приостанавливает работу на время загрузки (при необходимости) и выполнения сценария. В сценарии подобного рода может использоваться метод `document.write()` для вставки текста во входной поток, и такой текст станет частью документа, когда синтаксический анализатор возобновит работу. В сценарии такого рода часто просто определяются функции и регистрируются обработчики событий для будущего применения, но можно обходить и манипулировать деревом документа в том виде, в каком оно существует в данный момент. То есть немодульный сценарий, который не имеет атрибута `async` или `defer`, может видеть собственный дескриптор `<script>` и содержимое документа, находящееся перед ним.

3. Когда синтаксический анализатор встречает элемент `<script>`, который имеет установленный атрибут `async`, он начинает загрузку текста сценария (и если сценарий является модулем, то также рекурсивно загружаются все зависимости сценария) и продолжает анализ документа. Сценарий будет выполнен как можно скорее после окончания его загрузки, но синтаксический анализатор не останавливает работу в ожидании

загрузки сценария. В асинхронных сценариях не должен использоваться метод `document.write()`. Они могут видеть собственные дескрипторы `<script>` и все содержимое документа, находящееся перед ними, и могут иметь или не иметь доступ к дополнительному содержимому документа.

4. Когда документ полностью проанализирован, значение свойства `document.readyState` изменяется на `interactive` (взаимодействует).
5. Любые сценарии, которые имеют установленный атрибут `defer` (наряду с любыми модульными сценариями, не имеющими атрибута `async`), выполняются в том порядке, в каком они появляются в документе. В это время также могут выполняться асинхронные сценарии. Отложенные сценарии имеют доступ к полному документу и в них не должен применяться метод `document.write()`.
6. Браузер инициирует событие `DOMContentLoaded` для объекта `Document`. Это знаменует переход от синхронного этапа выполнения сценариев к асинхронному управляемому событиями этапу выполнения программы. Тем не менее, имейте в виду, что все еще могут существовать асинхронные сценарии, которые в данной точке пока не выполнены.
7. В этот момент документ полностью проанализирован, но браузер по-прежнему может ожидать загрузки дополнительного содержимого, такого как изображения. Когда загрузка всего содержимого такого рода завершена, а все асинхронные сценарии загружены и выполнены, значение свойства `document.readyState` изменяется на `complete` (укомплектован) и веб-браузер инициирует событие `load` для объекта `Window`.
8. С этого момента обработчики событий вызываются асинхронно в ответ на события пользовательского ввода, события сети, события истечения таймера и т.д.

15.1.6. Ввод и вывод программы

Как и любая программа, программа JavaScript стороны клиента обрабатывает входные данные, чтобы выпустить выходные данные. Доступны описанные ниже источники входных данных.

- Содержимое самого документа, доступ к которому код JavaScript может получать через API-интерфейс DOM (см. раздел 15.3).
- Пользовательский ввод в форме событий, такой как щелчки кнопками мыши (или касания сенсорного экрана) на HTML-элементах `<button>` или текст, введенный в HTML-элементах `<textarea>`. В разделе 15.2 будет показано, каким образом программы JavaScript могут реагировать на пользовательские события вроде упомянутых.
- URL отображаемого документа доступен в коде JavaScript стороны клиента как `document.URL`. Если вы передадите эту строку конструктору `URL()` (см. раздел 11.9), то легко сможете получить доступ к частям пути, запроса и фрагмента URL.

- Содержимое HTTP-заголовка Cookie запроса доступно в коде JavaScript стороны клиента как document.cookie. Cookie-наборы обычно используются кодом стороны сервера для поддержки пользовательских сеансов, но код стороны клиента при необходимости также может их читать (и записывать). Дополнительные детали ищите в разделе 15.12.2.
- Глобальный объект navigator предоставляет доступ к информации о веб-браузере, об операционной системе, под управлением которой он функционирует, и об их возможностях. Например, navigator.userAgent — это строка, идентифицирующая веб-браузер, navigator.language — предпочитаемый пользователем язык и navigator.hardwareConcurrency — количество логических центральных процессоров, доступных веб-браузеру. Аналогично глобальный объект screen предоставляет доступ к размеру дисплея пользователя через свойства screen.width и screen.height. В некотором смысле объекты navigator и screen являются для веб-браузеров тем же, чем переменные среды для программ Node.

Код JavaScript стороны клиента обычно выпускает вывод, когда необходимо, за счет манипулирования HTML-документом с помощью API-интерфейса DOM (см. раздел 15.3) или применения высокоуровневого фреймворка, подобного React или Angular, для манипулирования документом. Для вырабатывания вывода код стороны клиента также может использовать console.log() и связанные методы (см. раздел 11.8). Но такой вывод будет виден только в консоли инструментов разработчика, поэтому он полезен во время отладки, но не в качестве вывода, видимого пользователю.

15.1.7. Ошибки в программе

В отличие от приложений (таких как приложения Node), которые выполняются непосредственно под управлением операционной системы, программы JavaScript в веб-браузере не могут по-настоящему "отказать". Если во время выполнения вашей программы JavaScript возникает исключение, а вы не предусмотрели оператор catch для его обработки, тогда в консоли инструментов разработчика отобразится сообщение об ошибке, но все зарегистрированные обработчики событий продолжат выполняться и реагировать на события.

Если вы хотите определить обработчик ошибок для вызова в качестве последнего средства при возникновении неперехваченного исключения такого рода, тогда установите свойство onerror объекта Window в функцию обработчика ошибок. Когда неперехваченное исключение распространится вверх по стеку вызовов до самого конца и сообщение об ошибке готово отобразиться в консоли инструментов разработчика, функция window.onerror будет вызвана с тремя строковыми аргументами. В первом аргументе window.onerror передается сообщение, описывающее ошибку. Второй аргумент — это строка, которая содержит URL кода JavaScript, вызвавшего ошибку. В третьем аргументе указывается номер строки внутри документа, где возникла ошибка. Если обработчик onerror возвращает true, то тем самым уведомляет браузер о том, что ошибка

обработана и нет необходимости в добавочном действии — другими словами, браузер не должен отображать собственное сообщение об ошибке.

Когда объект `Promise` отклоняется и отсутствует функция `.catch()` для его обработки, возникает ситуация, во многом похожая на необработанное исключение: непредвиденная или логическая ошибка в вашей программе. Вы можете обнаружить это, определив функцию `window.onunhandledrejection` или применив `window.addEventListener()` для регистрации обработчика событий "необработанных отклонений". Передаваемый такому обработчику объект события будет иметь свойство `promise`, значением которого является отклоненный объект `Promise`, и свойство `reason` со значением, которое передавалось бы функции `.catch()`. Как и в случае описанных ранее обработчиков ошибок, если вы вызовите `preventDefault()` на объекте события необработанного отклонения, то оно будет считаться обработанным и не приведет к отображению сообщения об ошибке в консоли инструментов разработчика.

Определять обработчики `onerror` или `onunhandledrejection` придется нечасто, но они могут оказаться полезными в качестве механизма удаленного измерения, если вы хотите сообщать серверу об ошибках на стороне клиента (скажем, используя функцию `fetch()` для выполнения HTTP-запроса `POST`). Такой подход позволит получать информацию о непредвиденных ошибках, которые возникали в браузерах пользователей.

15.1.8. Модель безопасности веб-сети

Тот факт, что веб-страницы могут выполнять произвольный код JavaScript на вашем персональном устройстве, влечет за собой четкие следствия в плане безопасности, и поставщики браузеров напряженно работают над достижением баланса между двумя конкурирующими целями:

- определение мощных API-интерфейсов стороны клиента, чтобы сделать возможными полезные веб-приложения;
- препятствование чтению либо изменению ваших данных, нарушению вашей конфиденциальности, мошенничеству в отношении вас или ненужной трате вашего времени со стороны вредоносного кода.

В последующих подразделах представлен краткий обзор ограничений и проблем, связанных с безопасностью, о которых вы как программист на JavaScript должны знать.

Чего не может делать JavaScript стороны клиента

Первая линия защиты веб-браузеров от вредоносного кода заключается в том, что они попросту не поддерживают определенные возможности. Например, JavaScript стороны клиента не предоставляет никаких способов записи или удаления произвольных файлов либо получения списка произвольных каталогов на клиентском компьютере. Это означает, что программа JavaScript не может удалять данные или внедрять вирусы.

Точно так же JavaScript стороны клиента не располагает универсальными сетевыми возможностями. Программа JavaScript стороны клиента может делать HTTP-запросы (см. подраздел 15.11.1). А еще один стандарт, известный как WebSocket (см. подраздел 15.11.3), определяет похожий на сокеты API-интерфейс для взаимодействия со специализированными серверами. Но ни один из таких API-интерфейсов не разрешает непосредственный доступ к более широкой сети. Универсальные клиенты и серверы Интернета не могут быть написаны на JavaScript стороны клиента.

Политика одинакового источника

Политика одинакового источника (the same-origin policy) — это обширное ограничение безопасности, касающееся того, с каким веб-содержимым может взаимодействовать код JavaScript. Она обычно вступает в игру, когда веб-страница включает элементы `<iframe>`. В таком случае политика одинакового источника управляет взаимодействием кода JavaScript в одном фрейме с содержимым других фреймов. В частности, сценарий может читать только свойства окон и документов, которые происходят из того же источника, что и документ, содержащий сценарий.

Источник документа определяется как протокол, хост и порт URL, откуда документ был загружен. Документы, загруженные из разных веб-серверов, имеют отличающиеся источники. Документы, загруженные через разные порты того же самого хоста, имеют отличающиеся источники. И документ, загруженный посредством протокола `http:`, имеет источник, отличающийся от источника документа, который загружен с помощью протокола `https:`, даже если они поступали из того же самого веб-сервера. Браузеры обычно трактуют каждый URL типа `file:` как отдельный источник, так что если вы трудитесь над программой, которая отображает более одного документа из того же самого сервера, то возможно не сумеете протестировать ее локально с применением URL типа `file:` и будете вынуждены запускать статический веб-сервер во время разработки.

Важно понимать, что источник самого сценария не имеет отношения к политике одинакового источника: имеет значение источник документа, в который встроен сценарий. Предположим, например, что сценарий, размещенный на хосте A, включен (с использованием свойства `src` элемента `<script>`) в веб-страницу, обслуживаемую хостом B. Источником данного сценария является хост B, и сценарий получает полный доступ к содержимому документа, куда он внедрен. Если документ имеет элемент `<iframe>`, который содержит второй документ из хоста B, тогда сценарий получает полный доступ и к содержимому второго документа. Но если документ верхнего уровня содержит еще один элемент `<iframe>`, отображающий документ из хоста C (или даже из хоста A), то политика одинакового источника вступает в силу и предотвращает доступ сценария к этому вложенному документу.

Политика одинакового источника также применяется к сценарным HTTP-запросам (см. подраздел 15.11.1). В коде JavaScript можно выполнять произвольные HTTP-запросы к веб-серверу, из которого был загружен содержащий доку-

мент, но сценариям не разрешено взаимодействовать с другими веб-серверами (если только такие веб-серверы не принимают участие в CORS, как будет описано далее).

Политика одинакового источника создает проблемы для крупных веб-сайтов, которые используют множество поддоменов. Скажем, сценарии с источником `orders.example.com` могут нуждаться в чтении свойств из документов в `example.com`. Чтобы поддерживать многодоменные веб-сайты подобного рода, сценарии могут изменять свой источник, устанавливая `document.domain` в суффикс домена.

Таким образом, сценарий с источником `https://orders.example.com` может изменить свой источник на `https://example.com` за счет установки `document.domain` в `"example.com"`. Но сценарий не может устанавливать `document.domain` в `"orders.example"`, `"ample.com"` или `"com"`.

Второй методикой ослабления политики одинакового источника является совместное использование ресурсов между разными источниками (cross-origin resource sharing — CORS), которое позволяет серверам решать, какие источники они готовы обслуживать. CORS расширяет HTTP новым заголовком запроса `Origin:` и новым заголовком ответа `Access-Control-Allow-Origin`. Методика CORS дает возможность серверам применять заголовок для явного перечисления источников, которые могут запрашивать файл, или использовать групповой символ и разрешать файлу быть запрошенным любым сайтом. Браузеры принимают во внимание заголовки CORS и в случае их отсутствия не ослабляют политику одинакового источника.

Межсайтовые сценарии

Межсайтовые сценарии (cross-site scripting — XSS) — это термин для обозначения категории проблем, связанных с безопасностью, когда атакующий внедряет HTML-дескрипторы в целевой веб-сайт. Программисты на JavaScript стороны клиента обязаны знать о межсайтовых сценариях и защищаться от них.

Веб-страница уязвима для межсайтовых сценариев, если она динамически генерирует содержимое документа и основывает такое содержимое на данных, отправленных пользователем, без предварительной "очистки" данных за счет удаления из них любых внедренных HTML-дескрипторов. В качестве тривиального примера рассмотрим следующую веб-страницу, которая применяет JavaScript для приветствия пользователя по имени:

```
<script>
let name = new URL(document.URL).searchParams.get("name");
document.querySelector('h1').innerHTML = "Hello " + name;
</script>
```

Показанный двухстрочный сценарий извлекает входные данные из параметра запроса `"name"`, указанного в URL документа. Затем он использует API-интерфейс DOM для внедрения HTML-строки в первый дескриптор `<h1>` внутри документа. Страница предназначена для вызова посредством URL вида:

```
http://www.example.com/greet.html?name=David
```

При таком использовании она отображает текст "Hello David". Но посмотрим, что происходит, когда страница вызывается с таким параметром запроса:

```
name=%3Cimg%20src=%22x.png%22%20onload=%22alert(%27hacked%27)%22/%3E
```

Когда параметры будут декодированы, URL вызовет внедрение в документ следующей HTML-разметки:

```
Hello <img src="x.png" onload="alert('hacked')"/>
```

После загрузки изображения выполняется строка кода JavaScript в атрибуте `onload`. Глобальная функция `alert()` отображает модальное диалоговое окно. Появление одиночного диалогового окна относительно безболезненно, но демонстрирует возможность выполнения на этом сайте произвольного кода, потому что он отображает неочищенную HTML-разметку.

Атаки межсайтовыми сценариями называются так из-за того, что в них вовлечено более одного сайта. Сайт B включает специально изготовленную ссылку (подобную той, что была в предыдущем примере) на сайт A. Если сайт B сумеет убедить пользователей щелкнуть на ссылке, тогда они попадут на сайт A, но этот сайт теперь будет выполнять код из сайта B. Такой код может повредить страницу или заставить ее неправильно работать. Более опасно то, что вредоносный код мог бы прочитать cookie-наборы, сохраненные сайтом A (возможно, номера счетов или другую персональную идентифицирующую информацию), и отправить полученные данные сайту B. Внедренный код мог бы даже отслеживать нажатые пользователем клавиши и отправлять такие данные сайту B.

В целом способ предотвращения атак XSS предусматривает удаление HTML-дескрипторов из любых ненадежных данных до их применения при создании содержимого динамического документа. Вы можете исправить приведенный ранее файл `greet.html`, заменив специальные символы HTML в ненадежной входной строке эквивалентными сущностями HTML:

```
name = name
    .replace(/&/g, "&amp;")
    .replace(/</g, "&lt;")
    .replace(/>/g, "&gt;")
    .replace(/"/g, "&quot;")
    .replace(/'/g, "&#x27;")
    .replace(/\//g, "&#x2F;")
```

Еще один подход к решению проблемы с атаками XSS заключается в структурировании веб-приложений, чтобы ненадежное содержимое всегда отображалось в элементе `<iframe>` с атрибутом `sandbox`, установленным для запрета выполнения сценариев и других возможностей.

Межсайтовые сценарии — это опасная уязвимость, корни которой уходят глубоко в архитектуру веб-сети. Ее стоит досконально понимать, но дальнейшее обсуждение выходит за рамки настоящей книги. Существует много онлайновых ресурсов, которые помогут вам защититься от межсайтовых сценариев.

15.2. События

Программы JavaScript стороны клиента используют асинхронную и управляемую событиями программную модель. При таком стиле программирования веб-браузер генерирует событие всякий раз, когда происходит что-то интересное в документе, в браузере, в каком-то элементе или в ассоциированном с ним объекте. Например, веб-браузер генерирует событие, когда он заканчивает загрузку документа, когда пользователь наводит указатель мыши на гиперссылку или когда пользователь нажимает клавишу на клавиатуре. Если приложение JavaScript интересует определенный тип событий, тогда оно может зарегистрировать одну или большее количество функций, которые должны вызываться при возникновении событий такого типа. Обратите внимание, что подход не уникален для программирования веб-приложений: все приложения с графическим пользовательским интерфейсом спроектированы подобным образом — они ожидают взаимодействия (т.е. ждут появления событий) и затем реагируют.

В коде JavaScript стороны клиента события могут происходить в любом элементе внутри HTML-документа и данный факт сделает модель событий веб-браузера более сложной, чем модель событий Node. Мы начнем раздел с ряда важных определений, которые помогут объяснить модель событий.

Тип события

Строка, которая указывает вид происшедшего события. Например, тип `"mousemove"` означает, что пользователь переместил указатель мыши. Тип `"keydown"` означает, что пользователь нажал клавишу на клавиатуре. Тип `"load"` означает, что завершена загрузка документа (или какого-то другого ресурса) из сети. Поскольку тип события — всего лишь строка, ее временами называют именем события, и мы действительно применяем это имя для идентификации вида события, о котором идет речь.

Цель события

Объект, в котором событие возникло или с которым оно ассоциировано. Когда мы говорим о событии, то должны указывать и тип, и цель, например, событие загрузки в объекте `Window` или событие щелчка в объекте `Element` типа `<button>`. В приложениях JavaScript стороны клиента самыми распространенными целями событий являются объекты `Window`, `Document` и `Element`, но некоторые события инициируются объектами других видов. Скажем, объект `Worker` (разновидность потока, раскрываемая в разделе 15.13) — цель для событий `"message"`, которые возникают, когда поток веб-воркера посылает сообщение главному потоку.

Обработчик событий или прослушиватель событий

Функция, которая обрабатывает или реагирует на событие[2]. Приложения регистрируют свои функции обработчиков событий в веб-браузере, указы-

[2] В ряде источников, включая спецификацию HTML, проводится формальное различие между обработчиками и прослушивателями на основе способа их регистрации. В книге мы будем обходиться с этими двумя терминами как с синонимами.

вая тип события и цель события. Когда событие указанного типа происходит в указанной цели, браузер вызывает функцию обработчика. Когда обработчики событий вызываются для какого-то объекта, мы говорим, что браузер "инициировал", "запустил" или "отправил" событие. Существует несколько способов регистрации обработчиков событий и подробности регистрации и вызова обработчиков объясняются в подразделах 15.2.2 и 15.2.3.

Объект события

Объект, который ассоциирован с определенным событием и содержит детали об этом событии. Объект события передается как параметр функции обработчика событий. Все объекты событий имеют свойство `type`, которое указывает тип события, и свойство `target`, задающее цель события. Каждый тип события определяет набор свойств для ассоциированного с ним объекта события. Например, объект, ассоциированный с событием мыши, включает координаты указателя мыши, а объект, ассоциированный с событием клавиатуры, содержит детали о нажатой клавише и удерживаемых в нажатом состоянии модифицирующих клавишах. Многие типы событий определяют только несколько стандартных свойств, подобных `type` и `target`, и не несут много другой информации. Для таких событий имеет значение простое возникновение события, а не связанные с ним детали.

Распространение событий

Процесс, с помощью которого браузер решает, какие объекты должны запускать обработчики событий. Для событий, относящихся к одиночному объекту, таких как событие `"load"` объекта `Window` или событие `"message"` объекта `Worker`, распространение не требуется. Но когда определенные виды событий происходят в элементах внутри HTML-документа, они распространяются или поднимаются словно "пузырьки" вверх по дереву документа. Если пользователь наводит указатель мыши на гиперссылку, сначала инициируется событие `"mousemove"` в элементе `<a>`, который определяет ссылку. Затем оно инициируется в содержащих элементах: возможно, в элементе `<p>`, элементе `<section>` и самом объекте `Document`. Иногда более удобно регистрировать единственный обработчик событий в объекте `Document` или другом контейнерном элементе, чем регистрировать обработчики для каждого индивидуального элемента, который вас интересует. Обработчик событий может останавливать распространение, чтобы оно не продолжало подниматься и не запускало обработчики в содержащих элементах. Обработчики делают это вызовом метода на объекте события. В еще одной форме распространения, известной как *захватывание событий*, специально зарегистрированные в контейнерных элементах обработчики имеют возможность перехватывать (или "захватывать") события до того, как они доберутся до своей фактической цели. Пузырьковый подъем и захватывание событий подробно рассматриваются в подразделе 15.2.4.

С некоторыми событиями связаны *стандартные действия*. Скажем, когда происходит событие щелчка на гиперссылке, стандартным действием для браузера будет следование по ссылке и загрузка новой страницы. Обработчики событий могут препятствовать такому стандартному действию, вызывая метод на объекте события. Иногда это называют "отменой" события; детали ищите в подразделе 15.2.5.

15.2.1. Категории событий

В JavaScript стороны клиента поддерживается настолько большое количество типов событий, что описать все типы в одной главе нет никакой возможности. Однако может быть удобно сгруппировать события в ряд общих категорий, чтобы проиллюстрировать масштаб и широкий спектр поддерживаемых событий.

Зависимые от устройства события ввода

Эти события напрямую привязаны к конкретному устройству ввода, такому как мышь или клавиатура. Сюда входят типы событий наподобие `"mousedown"`, `"mousemove"`, `"mouseup"`, `"touchstart"`, `"touchmove"`, `"touchend"`, `"keydown"` и `"keyup"`.

Независимые от устройства события ввода

Такие события не привязаны напрямую к конкретному устройству ввода. Например, событие `"click"` указывает, что была активизирована ссылка или кнопка (либо другой элемент документа). Это часто делается через щелчок кнопкой мыши, но можно использовать и нажатие клавиши на клавиатуре или касание (на устройствах с сенсорными экранами). Событие `"input"` является независимой от устройства альтернативой событию `"keydown"` и поддерживает клавиатурный ввод, а также варианты вроде вырезания и вставки и методы ввода, применяемые при идеографическом письме. Типы событий `"pointerdown"`, `"pointermove"` и `"pointerup"` представляют собой независимые от устройства альтернативы событиям мыши и касания. Они также работают для указателей мыши, сенсорных экранов и перьевого ввода.

События пользовательского интерфейса

События пользовательского интерфейса — это события более высокого уровня, часто возникающие в элементах HTML-формы, которая определяет пользовательский интерфейс для веб-приложения. В их число входят событие `"focus"` (когда поле ввода текста получает клавиатурный фокус), событие `"change"` (когда пользователь изменяет значение, отображаемое элементом формы) и событие `"submit"` (когда пользователь щелкает на кнопке отправки формы).

События изменения состояния

Некоторые события инициируются не непосредственными действиями пользователя, а активностью сети или браузера, и указывают на какое-то изменение, связанное с жизненным циклом или состоянием. Вероятно, наиболее часто используемыми событиями такого рода являются `"load"` и `"DOMContentLoaded"`, возникающие в конце загрузки документа в объектах `Window` и `Document` соответственно (см. раздел "Временная шкала JavaScript стороны клиента" ранее в главе). Браузеры инициируют события `"online"` и `"offline"` в объекте `Window` при изменении подключаемости к сети. Механизм управления хронологией браузера (см. подраздел 15.10.4) инициирует событие `"popstate"` как ответ на щелчок на кнопке перехода на предыдущую страницу.

События, специфичные для API-интерфейса

Некоторые API-интерфейсы, определенные спецификацией HTML и связанными спецификациями, включают собственные типы событий. HTML-элементы `<video>` и `<audio>` определяют длинный список ассоциированных типов событий, таких как `"waiting"`, `"playing"`, `"seeking"`, `"volumechange"` и т.д., которые вы можете применять для настройки воспроизведения мультимедийного содержимого. В сущности, API-интерфейсы веб-платформы, которые являются асинхронными и были разработаны до появления в JavaScript объектов `Promise`, основаны на событиях и определяют события, специфичные для API-интерфейсов. Скажем, API-интерфейс IndexedDB (см. подраздел 15.12.3) инициирует события `"success"` и `"error"`, когда запросы к базе данных завершаются успешно или терпят неудачу. И хотя новый API-интерфейс `fetch()` (см. подраздел 15.11.1) для выполнения HTTP-запросов основан на `Promise`, в API-интерфейсе `XMLHttpRequest`, который он заменил, определено несколько типов событий, специфичных для API-интерфейса.

15.2.2. Регистрация обработчиков событий

Обработчики событий регистрируются двумя способами. Первый способ, доступный с ранней поры существования веб-сети, предусматривает установку свойства объекта или элемента документа, который является целью события. Второй способ (более новый и универсальный) заключается в передаче обработчика методу `addEventListener()` объекта или элемента.

Установка свойств для обработчиков событий

Простейший способ регистрации обработчика событий — установка свойства цели события в желаемую функцию обработчика событий. По соглашению свойства для обработчиков событий имеют имена, состоящие из слова `on`, за которым следует имя события: `onclick`, `onchange`, `onload`, `onmouseover` и т.д. Обратите внимание, что имена этих свойств чувствительны к регистру и запи-

сываются полностью в нижнем регистре[3], даже когда тип события (наподобие "mousedown") образован из нескольких слов. Следующий код включает две регистрации обработчиков событий такого рода:

```
// Установить свойство onload объекта Window в функцию. Функция
// является обработчиком событий: вызывается при загрузке документа.
window.onload = function() {
    // Искать элемент <form>.
    let form = document.querySelector("form#shipping");
    // Зарегистрировать функцию обработчика событий для формы,
    // которая будет вызываться до отправки формы. Предполагается,
    // что функция isFormValid() определена в другом месте.
    form.onsubmit = function(event) { // Когда пользователь
                                      // отправляет форму,
        if (!isFormValid(this)) {     // проверить, допустимы ли
                                      // входные данные формы,
            event.preventDefault();   // и если нет, тогда
                                      // воспрепятствовать отправке.
        }
    };
};
```

Недостаток свойств для обработчиков событий связан с тем, что они проектировались в предположении наличия у целей событий не более одного обработчика для каждого типа событий. Часто обработчики событий лучше регистрировать с использованием метода `addEventListener()`, потому что такой способ не переписывает любые ранее зарегистрированные обработчики.

Установка атрибутов для обработчиков событий

Свойства для обработчиков событий в элементах документа также можно определять прямо в HTML-файле как атрибуты соответствующих HTML-дескрипторах. (Обработчики, которые регистрировали бы для элемента `Window` в коде JavaScript, могут быть определены с помощью атрибутов дескриптора `<body>` в HTML-разметке.) В современной разработке веб-приложений такая методика обычно не приветствуется, но она возможна и документируется здесь, поскольку все еще встречается в существующем коде.

При определении обработчика событий как атрибута HTML-дескриптора значением атрибута должна быть строка кода JavaScript. Этот код обязан быть телом функции обработчика событий, а не полным объявлением функции. То есть код обработчика событий в атрибуте HTML-дескриптора не должен быть окружен фигурными скобками и не должен предваряться ключевым словом `function`. Вот пример:

```
<button onclick="console.log('Thank you');">Please Click</button>
```

[3] Если вы использовали фреймворк React для создания пользовательских интерфейсов стороны клиента, тогда это может вас удивить. Фреймворк React вносит ряд незначительных изменений в модель событий стороны клиента, и одно из них связано с тем, что имена свойств для обработчиков событий в React записываются в "верблюжьем" стиле: `onClick`, `onMouseOver` и т.д. Тем не менее, при работе с самой веб-платформой имена свойств для обработчиков событий записываются целиком в нижнем регистре.

Если обработчик событий в атрибуте HTML-дескриптора содержит несколько операторов JavaScript, тогда не забудьте отделить их друг от друга или разбить значение атрибута на несколько строк.

Когда вы указываете строку кода JavaScript в качестве значения атрибута для обработчика событий в HTML-дескрипторе, браузер преобразует вашу строку в функцию, которая выглядит подобно показанной ниже функции:

```
function(event) {
   with(document) {
      with(this.form || {}) {
         with(this) {
            /* ваш код */
         }
      }
   }
}
```

Аргумент `event` означает, что в коде вашего обработчика можно ссылаться на текущий объект события как на `event`. Операторы `with` означают, что в коде вашего обработчика можно ссылаться на свойства объекта цели, вмещающего элемента `<form>` (при его наличии) и вмещающего объекта `Document` напрямую, как если бы они были переменными в области видимости. Оператор `with` запрещен в строгом режиме (см. подраздел 5.6.3), но код JavaScript в атрибутах HTML-дескрипторов никогда не бывает строгим. Обработчики событий, определенные подобным способом, выполняются в среде, в которой определяются непредсказуемые переменные. В итоге могут возникать сбивающие с толку ошибки, что служит веской причиной избегать написания обработчиков событий в HTML-разметке.

`addEventListener()`

Любой объект, который может быть целью события, в том числе объекты `Window` и `Document` и все объекты `Element` документа, определяет метод по имени `addEventListener()`, с помощью которого можно регистрировать обработчик событий для данной цели. Метод `addEventListener()` принимает три аргумента. Первый аргумент — тип события, для которого регистрируется обработчик. Тип (или имя) события представляет собой строку, которая не включает префикс `on`, применяемый при установке свойств для обработчиков событий. Вторым аргументом `addEventListener()` является функция, подлежащая вызову, когда возникает событие указанного типа. Третий аргумент необязателен и объясняется ниже.

В следующем коде регистрируются два обработчика для события `"click"` в элементе `<button>`. Обратите внимание на отличия между двумя используемыми методиками:

```
<button id="mybutton">Click me</button>
<script>
let b = document.querySelector("#mybutton");
b.onclick = function() { console.log("Thanks for clicking me!"); };
b.addEventListener("click", () => { console.log("Thanks again!"); });
</script>
```

Вызов addEventListener() с "click" в первом аргументе не влияет на значение свойства onclick. В приведенном коде щелчок на кнопке будет выводить на консоль инструментов разработчика два сообщения. И если бы мы вызвали сначала addEventListener() и затем установили onclick, то все равно выводили бы на консоль два сообщения, только в обратном порядке. Что более важно, вы можете вызывать addEventListener() много раз, чтобы регистрировать более одной функции обработчика для того же самого типа события в том же самом объекте. Когда в объекте возникает событие, вызываются все обработчики, зарегистрированные для этого типа событий, в порядке, в котором они регистрировались. Вызов addEventListener() более одного раза для того же самого объекта с теми же самыми аргументами не оказывает никакого воздействия — функция обработчика остается зарегистрированной только однократно, а повторный вызов не изменяет порядок, в котором будут вызываться обработчики.

Метод addEventListener() связан с методом removeEventListener(), который ожидает такие же два аргумента (плюс необязательный третий), но вместо добавления удаляет функцию обработчика событий из объекта. Часто удобно временно зарегистрировать обработчик событий и вскоре после этого удалить его. Скажем, когда вы получили событие "mousedown", то можете временно зарегистрировать обработчики для событий "mousemove" и "mouseup", чтобы можно было видеть, перемещает ли пользователь указатель мыши. Затем при поступлении события "mouseup" вы отменяете регистрацию обработчиков для "mousemove" и "mouseup". В такой ситуации код удаления обработчиков событий может выглядеть следующим образом:

```
document.removeEventListener("mousemove", handleMouseMove);
document.removeEventListener("mouseup", handleMouseUp);
```

Необязательный третий аргумент метода addEventListener() представляет собой булевское значение или объект. Если вы передаете true, то функция обработчика регистрируется как *захватывающий* обработчик событий и вызывается на другой стадии передачи события. Захват событий будет раскрыт в подразделе 15.2.4. Если вы передаете true в третьем аргументе при регистрации прослушивателя событий, то обязаны также передать true в третьем аргументе метода removeEventListener(), когда хотите удалить обработчик.

Регистрация захватывающего обработчика событий — лишь один из трех вариантов, поддерживаемых методом addEventListener(), и вместо передачи одиночного булевского значения вы также можете передать объект, который явно указывает желаемые варианты:

```
document.addEventListener("click", handleClick, {
    capture: true,
    once: true,
    passive: true
});
```

Если объект Options имеет свойство capture, установленное в true, тогда обработчик событий будет зарегистрирован как захватывающий обработчик.

Если свойство capture, установлено в false или опущено, тогда обработчик не будет захватывающим.

Если объект Options имеет свойство once, установленное в true, тогда прослушиватель событий будет автоматически удален после однократного запуска. Если свойство once установлено в false или опущено, тогда обработчик никогда не удалится автоматически.

Если объект Options имеет свойство passive, установленное в true, то он указывает, что обработчик событий никогда не будет вызывать метод preventDefault() для отмены стандартного действия (см. подраздел 15.2.5). Это особенно важно для событий касания на мобильных устройствах — если обработчики событий для событий "touchmove" смогут препятствовать выполнению стандартного действия прокрутки браузера, тогда браузер не сумеет реализовать плавную прокрутку. Свойство passive предлагает способ регистрации потенциально разрушительного обработчика событий подобного рода, но позволяет веб-браузеру знать, что он может безопасно начать свою стандартную линию поведения вроде прокрутки во время выполнения обработчика событий. Плавная прокрутка настолько важна для удобного взаимодействия с пользователем, что браузеры Firefox и Chrome делают события "touchmove" и "mousewheel" пассивными по умолчанию. Таким образом, если вы действительно хотите зарегистрировать обработчик, который вызывает preventDefault() для одного из этих событий, тогда должны явно установить свойство passive в false.

Вы также можете передать объект Options методу removeEventListener(), но свойство capture — единственное, что имеет значение. При удалении прослушивателя нет необходимости указывать свойство once или passive, так что они игнорируются.

15.2.3. Вызов обработчиков событий

После регистрации обработчика событий веб-браузер будет вызывать его автоматически, когда в указанном объекте возникает событие указанного типа. В текущем подразделе подробно рассматривается вызов обработчиков событий с объяснением их аргумента, контекста вызова (значения this) и смысла возвращаемого значения.

Аргумент обработчика событий

Обработчики событий вызываются с объектом Event в качестве единственного аргумента. Свойства объекта Event предоставляют детали о событии.

- **type**. Тип события, которое произошло.
- **target**. Объект, в котором произошло событие.
- **currentTarget**. Для распространяемых событий это свойство представляет собой объект, в котором был зарегистрирован текущий обработчик событий.

- **timeStamp**. Отметка времени (в миллисекундах), которая представляет абсолютное время. Вы можете определить время, прошедшее между двумя событиями, путем вычитания отметки времени первого события из отметки времени второго события.
- **isTrusted**. Это свойство будет равно `true`, если событие было отправлено самим веб-браузером, и `false`, если кодом JavaScript.

Определенные виды событий имеют дополнительные свойства. Например, события мыши и указателя располагают свойствами `clientX` и `clientY`, которые указывают координаты окна, где событие произошло.

Контекст обработчика событий

Когда вы регистрируете обработчик событий, устанавливая свойство, ситуация выглядит так, будто вы определяете новый метод в объекте цели:

```
target.onclick = function() { /* код обработчика */ };
```

Поэтому неудивительно, что обработчики событий вызываются как методы объекта, в котором они определены. То есть внутри тела обработчика событий ключевое слово `this` ссылается на объект, для которого обработчик был зарегистрирован.

Обработчики вызываются с целью как значением `this`, даже при регистрации с применением метода `addEventListener()`. Однако это не так для обработчиков, определенных в виде стрелочных функций: стрелочные функции всегда имеют то же значение `this`, что и область видимости, в которой они определены.

Возвращаемое значение обработчика событий

В современном JavaScript обработчики событий ничего не должны возвращать. Вы могли видеть обработчики событий, которые возвращают значения, в более старом коде, и возвращаемое значение — это обычно сигнал браузеру о том, что он не должен выполнять стандартное действие, ассоциированное с событием. Скажем, если обработчик `onclick` кнопки отправки в форме возвращает `false`, тогда веб-браузер не будет отправлять форму (как правило, поскольку обработчик событий установил, что пользовательский ввод не прошел проверку допустимости на стороне клиента).

Стандартный и предпочтительный способ запретить браузеру выполнить стандартное действие предусматривает вызов метода `preventDefault()` (см. подраздел 15.2.5) на объекте `Event`.

Порядок вызова

Цель события может иметь несколько обработчиков событий, зарегистрированных для специфического типа событий. Когда возникает событие такого типа, браузер вызывает все обработчики в порядке, в котором они регистрировались. Интересно отметить, что это верно, даже если вы смешиваете обработчики событий, зарегистрированные посредством `addEventListener()`, с обработчиками событий, зарегистрированными через свойства объекта вроде `onclick`.

15.2.4. Распространение событий

Когда целью события является объект Window или другой автономный объект, браузер реагирует на событие, просто вызывая подходящие обработчики для такого одного объекта. Тем не менее, когда цель события — объект Document или объект Element документа, ситуация становится сложнее.

После того, как обработчики событий, зарегистрированные для элемента цели, вызваны, большинство событий поднимаются подобно пузырькам вверх по дереву DOM. Вызываются обработчики событий родителя цели. Затем вызываются обработчики, зарегистрированные для прародителя цели. Процесс продолжается вплоть до объекта Document и далее до объекта Window. Пузырьковый подъем предоставляет альтернативу регистрации обработчиков для множества индивидуальных элементов документа: взамен вы можете зарегистрировать единственный обработчик для элемента общего предка и обрабатывать в нем события. Например, вы можете зарегистрировать обработчик событий "change" для элемента <form> вместо того, чтобы регистрировать по одному обработчику событий "change" для каждого элемента формы.

Пузырьковый подъем характерен для большинства событий, возникающих в элементах документа. Заметные исключения — события "focus", "blur" и "scroll". Событие "load", происходящее в элементах документа, поднимается подобно пузырьку, но перестает всплывать в объекте Document и не распространяется до объекта Window. (Обработчик событий "load" объекта Window запускается, только когда документ полностью загружен.)

Пузырьковый подъем представляет собой вторую "стадию" распространения событий. Вызов обработчиков событий на самом объекте цели является второй стадией. Первая стадия, которая происходит еще до вызова обработчиков событий, называется стадией "захвата". Вспомните, что метод addEventListener() принимает необязательный третий аргумент. Если в этом аргументе передается true или {capture:true}, тогда обработчик регистрируется как захватывающий обработчик событий для вызова во время первой стадии распространения событий. Стадия захвата распространения событий похожа на стадию пузырькового подъема наоборот. Сначала вызываются захватывающие обработчики объекта Window, затем захватывающие обработчики объекта Document, затем объекта тела и так далее вниз по дереву DOM до тех пор, пока не будут вызваны захватывающие обработчики событий в родительском объекте цели события. Захватывающие обработчики событий, зарегистрированные в самой цели события, не вызываются.

Захватывание событий дает возможность взглянуть на события до того, как они доберутся до своей цели. Захватывающий обработчик событий может использоваться для отладки или применяться вместе с методикой отмены, описанной в следующем подразделе, для фильтрации событий, чтобы обработчики событий цели фактически не вызывались. Захватывание событий часто используется для обработки перетаскивания с помощью мыши, где события перемещения мыши должны обрабатываться перетаскиваемым объектом, а не элементами документа, над которыми он перетаскивается.

15.2.5. Отмена событий

Браузеры реагируют на многие пользовательские события, даже если ваш код не реагирует на них: когда пользователь щелкает кнопкой мыши на гиперссылке, браузер следует по ссылке. Если HTML-элемент для ввода текста имеет клавиатурный фокус и пользователь нажимает клавишу, то браузер вносит ввод пользователя. Если пользователь проводит пальцем по сенсорному экрану устройства, тогда браузер выполняет прокрутку. Если вы зарегистрируете обработчик для событий такого рода, то сможете запретить браузеру выполнять их стандартные действия, вызывая метод `preventDefault()` объекта события. (За исключением случая регистрации обработчика с объектом `Options`, имеющим установленное свойство `passive`, что делает вызов `preventDefault()` безрезультатным.)

Отмена стандартного действия, ассоциированного с событием, является лишь одним видом отмены событий. Мы можем также отменять распространение событий, вызывая метод `stopPropagation()` объекта события. Если для того же самого объекта определены другие обработчики, то оставшиеся обработчики по-прежнему будут вызваны, но после вызова `stopPropagation()` никакие прочие обработчики событий для любого другого объекта не вызываются. Метод `stopPropagation()` работает во время стадии захвата, в самой цели события и в течение стадии пузырькового подъема. Метод `stopImmediatePropagation()` работает подобно `stopPropagation()`, но также предотвращает вызов любых последующих обработчиков событий, зарегистрированных для того же объекта.

15.2.6. Отправка специальных событий

API-интерфейс для работы с событиями в коде JavaScript стороны клиента обладает относительно большой мощностью и позволяет вам определять и отправлять собственные события. Предположим, например, что вашей программе периодически требуется выполнять длительный расчет или делать сетевой запрос, и до тех пор, пока эта операция не закончится, другие операции невозможны. Вы хотите, чтобы пользователь знал об этом, путем отображения "вращателей" для указания о том, что приложение занято. Но занятый модуль не обязан знать, где должны отображаться вращатели. Взамен такой модуль мог бы просто отправить событие, чтобы объявить о своей занятости, и затем отправить еще одно событие, когда он больше не занят. Далее модуль пользовательского интерфейса может зарегистрировать обработчики для этих событий и предпринимать надлежащие действия, направленные на уведомление пользователя.

Если объект JavaScript имеет метод `addEventListener()`, тогда он будет "целью события", а значит иметь также и метод `dispatchEvent()`. Вы можете создать собственный объект события с помощью конструктора `CustomEvent()` и передать его методу `dispatchEvent()`. В первом аргументе `CustomEvent()` передается строка, указывающая тип вашего события, а во втором — объект, который задает свойства объекта события. Установите свойство `detail` этого объекта в строку, объект или другое значение, которое представляет содержимое вашего события. Если вы планируете отправлять свое событие элементу документа и хотите, чтобы оно поднималось подобно пузырьку вверх по дереву документа, тогда добавьте `bubbles:true` ко второму аргументу:

```
// Отправить специальное событие, чтобы уведомить
// пользовательский интерфейс о том, что мы заняты.
document.dispatchEvent(new CustomEvent("busy", { detail: true }));

// Выполнить сетевую операцию.
fetch(url)
   .then(handleNetworkResponse)
   .catch(handleNetworkError)
   .finally(() => {
      // После того, как сетевой запрос пройдет успешно или потерпит
      // неудачу, отправить еще одно событие, чтобы уведомить
      // пользовательский интерфейс о том, что мы больше не заняты.
      document.dispatchEvent(new CustomEvent("busy",
                                             { detail: false }));
   });

// Где-то в другом месте программы можно зарегистрировать обработчик
// для событий "busy" и использовать его для показа или сокрытия
// вращателя, уведомляющего пользователя о занятости.
document.addEventListener("busy", (e) => {
   if (e.detail) {
      showSpinner();
   } else {
      hideSpinner();
   }
});
```

15.3. Работа с документами в сценариях

JavaScript стороны клиента существует для того, чтобы превратить статические HTML-документы в интерактивные веб-приложения. Таким образом, снабжение содержимого веб-страниц сценариями действительно является основной целью JavaScript.

Каждый объект Window имеет свойство document, которое ссылается на объект Document. Объект Document представляет содержимое окна и рассматривается в настоящем разделе. Однако объект Document не является автономным. Это основной объект в модели DOM, предназначенный для представления и манипулирования содержимым документа.

Модель DOM была введена в подразделе 15.1.2. Здесь приводятся детальные объяснения API-интерфейса. Вы узнаете:

- как запрашивать или *выбирать* индивидуальные элементы из документа;
- каким образом делать *обход* документа и как находить предков, родственные элементы и потомков любого элемента документа;
- как запрашивать и устанавливать атрибуты элементов документа;
- каким образом запрашивать, устанавливать и модифицировать содержимое документа;
- как модифицировать структуру документа за счет создания, вставки и удаления узлов.

15.3.1. Выбор элементов документа

В программах JavaScript стороны клиента часто необходимо манипулировать одним или большим количеством элементов внутри документа. Глобальное свойство `document` ссылается на объект `Document`, а объект `Document` имеет свойства `head` и `body`, которые ссылаются на объекты `Element` для дескрипторов `<head>` и `<body>` соответственно. Но программа, где нужно манипулировать элементом, встроенным более глубоко в документ, должна каким-то образом получать или выбирать объекты `Element`, которые связаны с такими элементами документа.

Выбор элементов с использованием селекторов CSS

Таблицы стилей CSS располагают очень мощным синтаксисом, известным как *селекторы*, для описания элементов или наборов элементов внутри документа. Методы `querySelector()` и `querySelectorAll()` в DOM позволяют находить элемент или элементы внутри документа, которые соответствуют указанному селектору CSS. Прежде чем раскрывать методы, мы предлагаем краткое руководство по синтаксису селекторов CSS.

Селекторы CSS могут описывать элементы по имени дескриптора, по значению их атрибута `id` или по словам в их атрибуте `class`:

```
div             // Любой элемент <div>
#nav            // Элемент с id="nav"
.warning        // Любой элемент с "warning" в его атрибуте class
```

Символ `#` применяется для сопоставления на основе атрибута `id`, а символ `.` — для сопоставления на основе атрибута `class`. Элементы также можно выбирать на основе более общих значений атрибутов:

```
p[lang="fr"]    // Абзац, написанный на французском: <p lang="fr">
*[name="x"]     // Любой элемент с атрибутом name="x"
```

Обратите внимание, что в приведенных примерах комбинируются селектор имени дескриптора (или групповой символ `*` имени дескриптора) и селектор атрибута. Возможны также более сложные комбинации:

```
span.fatal.error        // Любой элемент <span> с "fatal"
                        // и "error" в его атрибуте class
span[lang="fr"].warning // Любой элемент <span> на французском
                        // с "warning" в его атрибуте class
```

Селекторы могут также задавать структуру документа:

```
#log span               // Любой потомок <span> элемента с id="log"
#log>span               // Любой дочерний <span> элемента с id="log"
body>h1:first-child     // Первый дочерний <h1> элемента <body>
img + p.caption         // Элемент <p> с "caption" в class,
                        // расположенный сразу после <img>
h2 ~ p                  // Любой элемент <p>, который следует после
                        // <h2> и является родственным ему
```

Если два селектора разделены запятой, то это значит, что мы выбираем элементы, которые соответствуют любому одному из селекторов:

```
button, input[type="button"]    // Все элементы <button>
                                // и <input type="button">
```

Как видите, селекторы CSS позволяют ссылаться на элементы документа по типу, идентификатору, классу, атрибутам и позиции внутри документа. Метод `querySelector()` принимает в качестве аргумента строку селектора CSS и возвращает первый совпадающий элемент в документе, который он находит, или `null`, если совпадений нет:

```
// Искать элемент документа для HTML-дескриптора
// с атрибутом id="spinner".
let spinner = document.querySelector("#spinner");
```

Метод `querySelectorAll()` аналогичен, но возвращает все совпадающие элементы в документе, а не только первый:

```
// Искать все объекты Element для дескрипторов <h1>, <h2> и <h3>.
let titles = document.querySelectorAll("h1, h2, h3");
```

Возвращаемое значение метода `querySelectorAll()` — это не массив объектов `Element`, а похожий на массив объект, известный как `NodeList`. Объекты `NodeList` имеют свойство `length` и могут индексироваться подобно массивам, так что по ним можно проходить посредством традиционного цикла `for`. Вдобавок объекты `NodeList` итерируемы и потому их можно использовать также с циклами `for/of`. Чтобы преобразовать `NodeList` в подлинный массив, его понадобится передать `Array.from()`.

Объект `NodeList`, возвращенный методом `querySelectorAll()`, будет иметь свойство `length`, установленное в 0, если в документе отсутствуют элементы, которые соответствовали бы указанному селектору.

Методы `querySelector()` и `querySelectorAll()` реализованы в классах `Element` и `Document`. В случае вызова на элементе они возвращают только элементы, являющиеся потомками данного элемента.

Обратите внимание, что в CSS определены псевдоэлементы `::first-line` и `::first-letter`. В CSS они соответствуют позициям текстовых узлов, а не фактическим элементам. Они не будут давать совпадения, когда применяются с методом `querySelectorAll()` или `querySelector()`. Кроме того, многие браузеры отказываются возвращать совпадения для псевдоклассов `:link` и `:visited`, т.к. это могло бы раскрыть информацию о хронологии просмотра пользователя.

Есть еще один метод выбора элементов на основе CSS — `closest()`. Он определен в классе `Element` и принимает в своем единственном аргументе селектор. Если селектор соответствует элементу, на котором вызван метод `closest()`, то этот элемент возвращается. Иначе возвращается ближайший предок, который соответствует селектору, или `null`, если совпадений не найдено. В некотором смысле `closest()` является противоположностью `querySelector()`: метод `closest()` начинает с элемента и ищет соответствие выше в дереве, в то время

как querySelector() начинает с элемента и ищет соответствие ниже в дереве. Метод closest() может быть полезен, когда вы регистрируете обработчик событий на высоком уровне в дереве документа. Если вы обрабатываете событие "click", например, то возможно желаете знать, выполнялся ли щелчок на гиперссылке. Объект события сообщит вам, какой была цель, но цель может оказаться текстом внутри ссылки, а не самим дескриптором <a> гиперссылки. Ваш обработчик события мог бы искать ближайшую содержащую гиперссылку:

```
//Найти ближайший объемлющий дескриптор <a>, который имеет атрибут href
let hyperlink = event.target.closest("a[href]");
```

Вот другой возможный способ использования closest():

```
// Возвратить true, если элемент e находится внутри
// спискового HTML-элемента.
function insideList(e) {
   return e.closest("ul,ol,dl") !== null;
}
```

Связанный метод matches() не возвращает предков или потомков: он просто проверяет, соответствует ли элемент селектору CSS, и в случае соответствия возвращает true, а иначе false:

```
// Возвратить true, если e - заголовочный HTML-элемент.
function isHeading(e) {
   return e.matches("h1,h2,h3,h4,h5,h6");
}
```

Другие методы выбора элементов

В дополнение к методам querySelector() и querySelectorAll() в DOM также определено несколько более старых методов выбора элементов, которые теперь в той или иной степени устарели. Но вы по-прежнему можете сталкиваться с их применением (особенно метода getElementById()):

```
// Искать элемент по идентификатору.
// Аргументом является только идентификатор без префикса #
// селектора CSS. Подобен document.querySelector("#sect1").
let sect1 = document.getElementById("sect1");

// Искать все элементы (такие как флажки в форме),
// которые имеют атрибут name="color".
// Подобен document.querySelectorAll('*[name="color"]').
let colors = document.getElementsByName("color");

// Искать все элементы <h1> в документе.
// Подобен document.querySelectorAll("h1").
let headings = document.getElementsByTagName("h1");

// Метод getElementsByTagName() также определен в элементах.
// Получить все элементы <h2> внутри элемента sect1.
let subheads = sect1.getElementsByTagName("h2");

// Искать все элементы, которые имеют класс "tooltip".
// Подобен document.querySelectorAll(".tooltip").
let tooltips = document.getElementsByClassName("tooltip");
```

```
// Искать всех потомков элемента sect1, которые имеют класс "sidebar".
// Подобен sect1.querySelectorAll(".sidebar").
let sidebars = sect1.getElementsByClassName("sidebar");
```

Как и `querySelectorAll()`, методы в приведенном выше коде возвращают `NodeList` (кроме `getElementById()`, который возвращает одиночный объект `Element`). Тем не менее, в отличие от `querySelectorAll()` объекты `NodeList`, возвращенные этими более старыми методами выбора элементов, являются "активными", т.е. длина и содержимое объектов `NodeList` может изменяться в случае изменения содержимого или структуры документа.

Предварительно выбранные элементы

По историческим причинам в классе `Document` определены сокращенные свойства для доступа к некоторым видам узлов. Скажем, свойства `images`, `forms` и `links` предоставляют легкий доступ к элементам `<img>`, `<form>` и `<a>` (но только к дескрипторам `<a>`, имеющим атрибут `href`) документа. Такие свойства ссылаются на объекты `HTMLCollection`, которые во многом похожи на объекты `NodeList`, но могут дополнительно индексироваться по идентификатору или по имени элемента. Например, с помощью свойства `document.forms` можно получить доступ к дескриптору `<form id="address">`:

```
document.forms.address;
```

Еще более устаревшим API-интерфейсом для выбора элементов является свойство `document.all`, которое похоже на объект `HTMLCollection` со всеми элементами в документе. Использовать свойство `document.all` больше не рекомендуется.

15.3.2. Структура и обход документа

После выбора объекта `Element` из `Document` иногда вам необходимо найти структурно связанные порции (родительский элемент, родственные элементы, дочерние элементы) документа. Когда нас в первую очередь интересуют объекты `Element` документа, а не текст внутри них (и пробельные символы между ними, также относящиеся к тексту), то имеется API-интерфейс обхода, который позволяет трактовать документ как дерево объектов `Element` и игнорировать узлы `Text`, входящие в состав документа. Этот API-интерфейс обхода не включает в себя методы; он представляет собой просто набор свойств в объектах `Element`, которые дают возможность ссылаться на родительский элемент, дочерние элементы и родственные элементы заданного элемента.

- **`parentNode`**. Это свойство элемента ссылается на родителя элемента, которым будет другой объект `Element` или объект `Document`.
- **`children`**. Этот объект `NodeList` содержит дочерние элементы типа `Element` элемента, но исключает дочерние элементы, отличающиеся от `Element`, вроде узлов `Text` (и узлов `Comment`).
- **`childElementCount`**. Количество дочерних элементов типа `Element`. Возвращает то же значение, что и `children.length`.

- **firstElementChild, lastElementChild**. Эти свойства ссылаются на первый и последний дочерние элементы типа Element элемента. Они будут равны null, если элемент не имеет дочерних элементов типа Element.
- **nextElementSibling, previousElementSibling**. Эти свойства ссылаются на родственные элементы типа Element непосредственно до или непосредственно после элемента или равны null, если родственных элементов нет.

С применением описанных выше свойств Element на второй дочерний элемент типа Element первого дочернего элемента объекта Document можно ссылаться посредством любого из следующих двух выражений:

```
document.children[0].children[1]
document.firstElementChild.firstElementChild.nextElementSibling
```

(В стандартном HTML-документе оба выражения ссылаются на дескриптор <body> документа.)

Ниже показаны две функции, которые демонстрируют, как вы можете использовать эти свойства для рекурсивного обхода в глубину документа с вызовом указанной функции для каждого элемента в документе:

```
// Выполняет рекурсивный обход объекта e типа Document или Element
// с вызовом функции f для объекта e и для каждого его потомка.
function traverse(e, f) {
    f(e);                                  // Вызвать f() для e
    for(let child of e.children) {         // Проход по дочерним элементам
        traverse(child, f);                // Рекурсия для каждого
    }
}
function traverse2(e, f) {
    f(e);                                  // Вызвать f() для e
    let child = e.firstElementChild; // Проход по дочерним элементам
                                           // в стиле связного списка
    while(child !== null) {
        traverse2(child, f);               // И рекурсия
        child = child.nextElementSibling;
    }
}
```

Документы как деревья узлов

Если вы просто хотите выполнить обход документа или какой-то его части, не игнорируя узлы Text, то можете применять другой набор свойств, определенных во всех объектах Node. Тогда вам удастся увидеть объекты Element, узлы Text и даже узлы Comment (представляющие HTML-комментарии в документе).

Во всех объектах Node определены следующие свойства.

- **parentNode**. Узел, который является родителем данного узла, или null для узлов наподобие объекта Document, не имеющим родителей.
- **childNodes**. Объект NodeList, допускающий только чтение, который содержит все дочерние узлы (не только дочерние объекты Element) данного узла.

- **firstChild, lastChild**. Первый и последний дочерние узлы данного узла или null, если дочерние узлы отсутствуют.
- **nextSibling, previousSibling**. Следующий и предыдущий родственные узлы данного узла. Эти свойства соединяют узлы в двусвязном списке.
- **nodeType**. Число, которое указывает, что это за узел. Узлы Document имеют значение 9, узлы Element — 1, узлы Text — 3, узлы Comment — 8.
- **nodeValue**. Текстовое содержимое узла Text или Comment.
- **nodeName**. Имя HTML-дескриптора объекта Element, преобразованное в верхний регистр.

С использованием перечисленных выше свойств Node ссылаться на второй дочерний узел первого дочернего узла объекта Document можно с помощью таких выражений:

```
document.childNodes[0].childNodes[1]
document.firstChild.firstChild.nextSibling
```

Предположим, что речь идет о документе следующего вида:

```
<html><head><title>Test</title></head><body>Hello World!</body></html>
```

Тогда вторым дочерним узлом первого дочернего узла будет элемент <body>. Его свойство nodeType имеет значение 1, а свойство nodeName — значение "BODY".

Однако обратите внимание, что этот API-интерфейс крайне чувствителен к изменениям в тексте документа. Скажем, если документ модифицируется путем вставки одиночного символа новой строки между дескрипторами <html> и <head>, то узел Text, который представляет символ новой строки, становится первым дочерним узлом первого дочернего узла, а вторым дочерним узлом будет элемент <head>, а не <body>.

В качестве демонстрации работы с API-интерфейсом обхода, основанного на Node, ниже приведена функция, которая возвращает весь текст внутри элемента или документа:

```
// Возвращает текстовое содержимое элемента e с рекурсией по дочерним
// элементам. Этот метод работает подобно свойству textContent.
function textContent(e) {
    let s = "";                          // Здесь накапливается текст.
    for(let child = e.firstChild; child !== null;
        child = child.nextSibling) {
        let type = child.nodeType;
        if (type === 3) {                // Если это узел Text,
            s += child.nodeValue;        // тогда добавить текстовое
                                         // содержимое к строке s.
        } else if (type === 1) {         // Если это узел Element,
            s += textContent(child);     // тогда организовать рекурсию.
        }
    }
    return s;
}
```

Функция представлена только для демонстрационных целей — на практике получить текстовое содержимое элемента e можно с помощью `e.textContent`.

15.3.3. Атрибуты

HTML-элементы состоят из имени дескриптора и набора пар имя/значение, известных как атрибуты. Например, элемент `<a>`, который определяет гиперссылку, применяет свой атрибут `href` как адресат ссылки.

В классе `Element` определены универсальные методы `getAttribute()`, `setAttribute()`, `hasAttribute()` и `removeAttribute()` для запрашивания, установки, проверки и удаления атрибутов элемента. Но значения атрибутов HTML-элементов (для всех стандартных атрибутов стандартных HTML-элементов) доступны в виде свойств объектов `HTMLElement`, которые представляют эти элементы, и обычно намного легче работать с ними как со свойствами JavaScript, чем вызывать `getAttribute()` и связанные методы.

Атрибуты HTML как свойства элементов

Объекты `Element`, представляющие элементы HTML-документа, в большинстве случаев определяют свойства для чтения/записи, которые отражают HTML-атрибуты элементов. В `Element` определены свойства для универсальных HTML-атрибутов, таких как `id`, `title`, `lang` и `dir`, а также свойства для обработчиков событий вроде `onclick`. В подтипах `Element` определены свойства, специфичные для конкретных элементов. Скажем, чтобы запросить URL изображения, можно использовать свойство `src` объекта `HTMLElement`, который представляет элемент `<img>`:

```
let image = document.querySelector("#main_image");
let url = image.src;          // Атрибут src - это URL изображения.
image.id === "main_image"     // => true; мы ищем изображение по id.
```

Аналогично с применением показанного ниже кода можно установить атрибуты отправки формы элемента `<form>`:

```
let f = document.querySelector("form");   // Первый элемент <form>
                                          // в документе.
f.action="https://www.example.com/submit";//Установить URL для отправки
f.method = "POST";                        //Установить тип HTTP-запроса
```

Для некоторых элементов, таких как `<input>`, некоторые HTML-атрибуты отображаются на свойства, именованные по-другому. Например, HTML-атрибут `value` элемента `<input>` отражается JavaScript-свойством `defaultValue`. JavaScript-свойство `value` элемента `<input>` содержит текущий пользовательский ввод, но изменения в свойстве `value` не влияют ни на свойство `defaultValue`, ни на атрибут `value`.

Атрибуты HTML не чувствительны к регистру, но свойства JavaScript чувствительны. Чтобы преобразовать имя атрибута в свойство JavaScript, запишите его в нижнем регистре. Однако если имя атрибута содержит более одного слова, тогда начинайте каждое слово после первого с буквы в верхнем регистре: ска-

жем, defaultChecked и tabIndex. Тем не менее, свойства для обработчиков событий наподобие onclick являются исключением и записываются в нижнем регистре.

Имена некоторых HTML-атрибутов относятся к зарезервированным словам в JavaScript. Общее правило в таком случае — снабжать имена свойств префиксом html. Например, HTML-атрибут for (элемента <label>) становится JavaScript-свойством htmlFor. В JavaScript слово class считается зарезервированным, а очень важный HTML-атрибут class является исключением из указанного правила: в коде JavaScript он становится свойством className.

Свойства, которые представляют HTML-атрибуты, обычно имеют строковые значения. Но когда атрибут имеет булевское или числовое значение (скажем, атрибуты defaultChecked и maxLength элемента <input>), свойства будут булевскими или числовыми, а не строковыми. Значениями атрибутов для обработчиков событий всегда будут функции (или null).

Обратите внимание, что в основанном на свойствах API-интерфейсе для получения и установки значений атрибутов не предусмотрено какого-либо способа удаления атрибута из элемента. В частности, использовать для такой цели операцию delete нельзя. При необходимости удаления атрибута применяйте метод removeAttribute().

Атрибут class

Атрибут class элемента HTML особенно важен. Его значением является разделенный пробелами список классов CSS, которые применяются к элементу и влияют на то, как он стилизуется посредством CSS. Поскольку class — зарезервированное слово в JavaScript, значение этого атрибута доступно через свойство className объектов Element. Свойство className может устанавливать и возвращать значение атрибута class в виде строки. Но атрибут class назван неудачно: его значение представляет собой список классов CSS, а не одиночный класс, и в коде JavaScript стороны клиента часто нужно добавлять и удалять отдельные имена классов из такого списка вместо того, чтобы работать со списком как с единой строкой.

По указанной причине в объектах Element определено свойство classList, которое позволяет обращаться с атрибутом class как со списком. Значением свойства classList является итерируемый объект, похожий на массив. Хотя свойство имеет имя classList, его поведение больше похоже на поведение множества классов с учетом того, что в нем определены методы add(), remove(), contains() и toggle():

```
// Когда мы хотим уведомить пользователя о том, что заняты,
// мы отображаем вращатель.
// Для этого мы должны удалить класс "hidden" и добавить класс "animated"
// (предполагается, что таблицы стилей сконфигурированы корректно).
let spinner = document.querySelector("#spinner");
spinner.classList.remove("hidden");
spinner.classList.add("animated");
```

Атрибуты набора данных

Временами удобно присоединять к HTML-элементам дополнительную информацию, как правило, когда код JavaScript будет выбирать такие элементы и каким-то образом манипулировать ими. В HTML считается допустимым любой атрибут с именем в нижнем регистре, начинающимся с префикса `data-`, и вы можете использовать их для любых целей. "Атрибуты набора данных" подобного рода не будут воздействовать на представление элементов, в котором они появляются, и определяют стандартный способ присоединения дополнительных данных, не нарушающий допустимость документа.

В модели DOM объекты `Element` имеют свойство `dataset`, относящееся к объекту, который имеет свойства, соответствующие атрибутам `data-` с удаленным префиксом `data-`. Следовательно, свойство `dataset.x` будет хранить значение атрибута `data-x`. Атрибуты, написанные через дефис, отображаются на имена свойств, записанные в "верблюжьем" стиле: атрибут `data-section-number` становится свойством `dataset.sectionNumber`.

Пусть у вас есть HTML-документ, содержащий такой текст:

```
<h2 id="title" data-section-number="16.1">Attributes</h2>
```

Тогда для доступа к номеру раздела вы могли бы написать следующий код JavaScript:

```
let number = document.querySelector("#title").dataset.sectionNumber;
```

15.3.4. Содержимое элементов

Взгляните еще раз на дерево документа, изображенное на рис. 15.1, и спросите себя, каково "содержимое" элемента <p>. Ответить на этот вопрос можно двумя путями:

- содержимым является HTML-строка "This is a simple document";
- содержимым является простая текстовая строка "This is a simple document".

Оба ответа правильны, и каждый ответ по-своему полезен. В последующих подразделах объясняется, как работать с представлениями содержимого элемента в виде HTML-разметки и простого текста.

Содержимое элементов в виде HTML-разметки

Чтение свойства `innerHTML` объекта `Element` приводит к возвращению содержимого этого элемента в виде строки разметки. Установка данного свойства инициирует вызов синтаксического анализатора веб-браузера и замещение текущего содержимого элемента проанализированным представлением новой строки. Вы можете проверить сказанное, открыв консоль инструментов разработчика и набрав такой код:

```
document.body.innerHTML = "<h1>Oops</h1>";
```

Вы увидите, что вся веб-страница исчезнет и будет заменена единственным заголовком "Oops". Веб-браузеры очень хороши в том, что касается синтаксического анализа HTML-разметки, и установка `innerHTML` обычно достаточно эф-

фективна. Однако имейте в виду, что дополнение свойства innerHTML текстом с помощью операции += не будет эффективным, т.к. требует шага сериализации для преобразования содержимого элемента в строку и затем шага синтаксического анализа для преобразования новой строки обратно в содержимое элемента.

При использовании этих API-интерфейсов для HTML очень важно никогда не вставлять в документ данные, введенные пользователем. Поступая так, вы открываете злоумышленникам возможность внедрения их собственных сценариев в ваше приложение. См. подраздел "Межсайтовые сценарии" ранее в главе.

Свойство outerHTML в Element похоже на innerHTML, но его значение включает сам элемент. Когда вы запрашиваете свойство outerHTML, значение включает открывающий и закрывающий дескрипторы элемента. А когда вы устанавливаете свойство outerHTML элемента, новое содержимое замещает сам элемент.

Связанный метод insertAdjacentHTML() в Element позволяет вставлять строку произвольной HTML-разметки "по соседству" с указанным элементом. Разметка передается методу во втором аргументе, а точный смысл "по соседству" зависит от значения первого аргумента, которым должна быть строка с одним из значений "beforebegin" (перед началом), "afterbegin" (после начала), "beforeend" (перед концом) или "afterend" (после конца). Перечисленные значения соответствуют точкам вставки, как иллюстрируется на рис. 15.2.

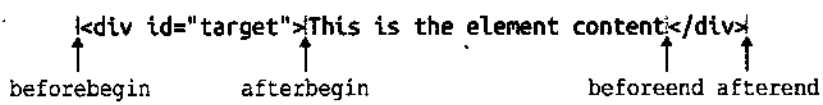


Рис. 15.2. Точки вставки для метода insertAdjacentHTML()

Содержимое элементов в виде простого текста

Иногда желательно запрашивать содержимое элемента в виде простого текста или вставлять в документ простой текст (без необходимости в отмене угловых скобок и амперсандов, применяемых в HTML-разметке). Стандартный способ предусматривает использование свойства textContent:

```
let para = document.querySelector("p"); // Первый элемент <p>
                                        // в документе.
let text = para.textContent;            // Получить текст абзаца.
para.textContent = "Hello World!";      // Изменить текст абзаца.
```

Свойство textContent определено в классе Node, поэтому оно работает для узлов Text и Element. В случае узлов Element оно находит и возвращает весь текст во всех потомках элемента.

В классе Element определено свойство innerText, которое похоже на textContent. Свойство innerText обладает необычным и сложным поведением, например, пытается предохранить табличное форматирование. Тем не менее, оно не определено как следует и не реализовано совместимым между браузерами образом, а потому больше не должно использоваться.

Текст в элементах <script>

Встроенные элементы <script> (т.е. без атрибута src) имеют свойство text, которое можно применять для извлечения их текста. Содержимое элемента <script> никогда не отображается браузером, а синтаксический анализатор HTML игнорирует угловые скобки и амперсанды внутри сценария. Это превращает элемент <script> в идеальное место для внедрения произвольных текстовых данных, используемых приложением. Просто установите атрибут type элемента в определенное значение (такое как "text/x-custom-data"), которое дает понять, что сценарий не является исполняемым кодом JavaScript. Тогда интерпретатор JavaScript проигнорирует сценарий, но элемент будет существовать в дереве документа, а его свойство text возвратит необходимые данные.

15.3.5. Создание, вставка и удаление узлов

Вы уже видели, как запрашивать и изменять содержимое документа с применением строк HTML-разметки и простого текста. И вы также видели, что мы можем обходить объект Document для исследования отдельных узлов Element и Text, из которых он состоит. Кроме того, документ можно изменять на уровне индивидуальных узлов. В классе Document определены методы для создания объектов Element, а объекты Element и Text имеют методы для вставки, удаления и замены узлов в дереве.

Создадим новый элемент с помощью метода createElement() класса Document и добавим в него строки текста или другие элементы посредством его методов append() и prepend():

```
let paragraph = document.createElement("p");// Создать пустой
                                            // элемент <p>.
let emphasis = document.createElement("em");// Создать пустой
                                            // элемент <em>.
emphasis.append("World");       // Добавить текст в элемент <em>.
paragraph.append("Hello ",emphasis,"!"); // Добавить текст и <em> в <p>
paragraph.prepend(";");         // Еще добавить текст в начало <p>.
paragraph.innerHTML             // => ";Hello <em>World</em>!"
```

Методы append() и prepend() принимают любое количество аргументов, которыми могут быть объекты Node или строки. Строковые аргументы автоматически преобразуются в узлы Text. (Вы можете создавать узлы Text явно с помощью document.createTextNode(), но для этого редко есть причина.) Метод append() добавляет к элементу аргументы в конец списка дочерних элементов. Метод prepend() добавляет аргументы в начало списка дочерних элементов.

Если вы хотите вставить узел Element или Text в середину списка дочерних элементов вмещающего элемента, тогда вам не подойдет ни append(), ни prepend(). В таком случае вы должны получить ссылку на родственный узел и вызвать метод before() для вставки нового содержимого до или метод after() для вставки нового содержимого после родственного узла. Вот пример:

```
// Найти заголовочный элемент с class="greetings".
let greetings = document.querySelector("h2.greetings");

// Вставить новый абзац и горизонтальную линию после этого
// заголовочного элемента.
greetings.after(paragraph, document.createElement("hr"));
```

Подобно `append()` и `prepend()` методы `after()` и `before()` принимают любое количество аргументов и после преобразования строк в узлы `Text` вставляют их всех в документ. Методы `append()` и `prepend()` определены только в объектах `Element`, но методы `after()` и `before()` работают с узлами `Element` и `Text`: вы можете использовать их для вставки содержимого относительно узла `Text`.

Обратите внимание, что элементы могут быть вставлены только в одном месте документа. Если элемент уже присутствует в документе, и вы вставляете его где-то в другом месте, тогда он будет перемещен в новое местоположение, а не скопирован:

```
// Мы вставили абзац после этого элемента, но теперь перемещаем его,
// так что он появляется перед элементом.
greetings.before(paragraph);
```

Если вы желаете сделать копию элемента, тогда применяйте метод `cloneNode()`, передавая ему `true` для копирования всего содержимого элемента:

```
// Создать копию абзаца и вставить ее после элемента с приветствием.
greetings.after(paragraph.cloneNode(true));
```

Вы можете удалить узел `Element` или `Text` из документа, вызвав метод `remove()`, или же заменить его, вызвав метод `replaceWith()`. Метод `remove()` не принимает аргументов, а `replaceWith()` принимает любое количество строк и элементов, в точности как делают методы `before()` и `after()`:

```
// Удалить элемент с приветствием из документа и заменить его
// абзацным элементом (перемещая абзац из текущего местоположения,
// если он уже был вставлен в документ).
greetings.replaceWith(paragraph);

// А теперь удалить абзац.
paragraph.remove();
```

В API-интерфейсе DOM также определено более старое поколение методов для вставки и удаления содержимого. Методы `appendChild()`, `insertBefore()`, `replaceChild()` и `removeChild()` труднее в использовании, чем обсуждаемые здесь методы, и никогда не понадобятся.

15.3.6. Пример: генерация оглавления

В примере 15.1 показано, как динамически создавать оглавление для документа. В нем демонстрируются многие методики работы с документами в сценариях, описанные в предшествующих подразделах. Пример хорошо прокомментирован и у вас не должны возникать сложности с пониманием кода.

Пример 15.1. Генерация оглавления с помощью API-интерфейса DOM

```
/**
 * TOC.js: создает оглавление для документа.
 *
 * Этот сценарий запускается, когда инициируется событие DOMContentLoaded,
 * и автоматически генерирует оглавление для документа.
 * Глобальные символы в нем не определяются, так что он не должен
 * конфликтовать с другими сценариями.
 *
 * После запуска этот сценарий сначала ищет элемент документа
 * с идентификатором "TOC". Если такого элемента нет, он создается
 * в начале документа. Далее функция ищет все дескрипторы от <h2>
 * до <h6>, трактует их как заголовки разделов и создает оглавление
 * внутри элемента TOC. Функция добавляет к заголовкам разделов
 * номера разделов и помещает заголовки внутрь именованных якорей,
 * чтобы TOC мог ссылаться на них. Сгенерированные якоря имеют имена,
 * которые начинаются с "TOC", поэтому в собственной HTML-разметке
 * вы должны избегать такого префикса.
 *
 * Записи в сгенерированном TOC могут быть стилизованы посредством CSS.
 * Все записи имеют класс "TOCEntry". Записи также имеют класс, который
 * соответствует уровню заголовка раздела. Дескрипторы <h1> генерируют
 * записи класса "TOCLevel1", дескрипторы <h2> - записи класса "TOCLevel2"
 * и т.д. Номера разделов, вставленные в заголовки, имеют класс "TOCSectNum".
 *
 * Вы можете использовать этот сценарий с таблицей стилей следующего вида:
 *
 *   #TOC { border: solid black 1px; margin: 10px; padding: 10px; }
 *   .TOCEntry { margin: 5px 0px; }
 *   .TOCEntry a { text-decoration: none; }
 *   .TOCLevel1 { font-size: 16pt; font-weight: bold; }
 *   .TOCLevel2 { font-size: 14pt; margin-left: .25in; }
 *   .TOCLevel3 { font-size: 12pt; margin-left: .5in; }
 *   .TOCSectNum:after { content: ": "; }
 *
 * Чтобы скрыть номера разделов, используйте такой стиль use this:
 *
 *   .TOCSectNum { display: none }
 **/
document.addEventListener("DOMContentLoaded", () => {
    // Найти контейнерный элемент TOC.
    // Если он отсутствует, тогда создать его в начале документа.
    let toc = document.querySelector("#TOC");
    if (!toc) {
        toc = document.createElement("div");
        toc.id = "TOC";
        document.body.prepend(toc);
    }

    // Найти все элементы заголовков разделов. Мы предполагаем, что заголовок
    // документа использует <h1>, а разделы внутри документа помечены с помощью
    // дескрипторов от <h2> до <h6>.
```

```
let headings = document.querySelectorAll("h2,h3,h4,h5,h6");

// Инициализировать массив, в котором отслеживаются номера разделов.
let sectionNumbers = [0,0,0,0,0];

// Пройти в цикле по найденным элементам заголовков разделов.
for(let heading of headings) {
   // Пропустить заголовок, если он присутствует внутри контейнера TOC.
   if (heading.parentNode === toc) {
      continue;
   }

   // Выяснить, какого уровня этот заголовок.
   // Вычесть 1, потому что <h2> - заголовок уровня level-1.
   let level = parseInt(heading.tagName.charAt(1)) - 1;

   // Инкрементировать номер раздела для этого уровня заголовка
   // и сбросить все более низкие номера уровней заголовков в 0.
   sectionNumbers[level-1]++;
   for(let i = level; i < sectionNumbers.length; i++) {
      sectionNumbers[i] = 0;
   }

   // Объединить номера разделов для всех уровней заголовков,
   // чтобы получить номер раздела вроде 2.3.1.
   let sectionNumber = sectionNumbers.slice(0, level).join(".");

   // Добавить номер раздела к заголовку раздела.
   // Мы помещаем номер в <span>, чтобы сделать его стилизуемым.
   let span = document.createElement("span");
   span.className = "TOCSectNum";
   span.textContent = sectionNumber;
   heading.prepend(span);

   // Поместить заголовок в именованный якорь,
   // чтобы на него можно было ссылаться.
   let anchor = document.createElement("a");
   let fragmentName = `TOC${sectionNumber}`;
   anchor.name = fragmentName;
   heading.before(anchor);  // Вставить якорь перед заголовком
   anchor.append(heading);  // и переместить заголовок внутрь якоря.

   // Создать ссылку на этот раздел.
   let link = document.createElement("a");
   link.href = `#${fragmentName}`;  // Адресат ссылки.

   // Копировать текст заголовка в ссылку. Это безопасное использование
   // innerHTML, т.к. мы не вставляем никаких ненадежных строк.
   link.innerHTML = heading.innerHTML;

   // Поместить ссылку в элемент div, который стилизуется на основе уровня.
   let entry = document.createElement("div");
   entry.classList.add("TOCEntry", `TOCLevel${level}`);
   entry.append(link);

   // Добавить элемент div в контейнер TOC.
   toc.append(entry);
}
```

15.4. Работа с CSS в сценариях

Вы видели, что в коде JavaScript можно управлять логической структурой и содержимым HTML-документов. Можно также управлять внешним видом и компоновкой HTML-документов, взаимодействуя в сценариях с CSS. В последующих подразделах объясняется несколько методик применения кода JavaScript для работы с CSS.

Книга посвящена языку JavaScript, а не CSS, и в этом разделе предполагается, что у вас уже есть практические навыки использования CSS для стилизации HTML-содержимого. Но стоит упомянуть о ряде стилей CSS, с которыми обычно взаимодействуют в коде JavaScript.

- Установка стиля `display` в `"none"` скрывает элемент. Позже элемент можно показать, установив `display` в какое-то другое значение.
- Элементы можно динамически позиционировать, устанавливая стиль `position` в `"absolute"`, `"relative"` или `"fixed"` и затем указывая в стилях `top` и `left` желаемые координаты. Такое позиционирование важно, когда JavaScript применяется для отображения динамического содержимого наподобие модельных диалоговых окон и всплывающих подсказок.
- Элементы можно сдвигать, масштабировать и вращать посредством стиля `transform`.
- С помощью стиля `transition` можно организовать анимацию изменений в других стилях CSS. Такая анимация автоматически обрабатывается веб-браузером и не требует JavaScript, но код JavaScript можно использовать для запуска анимации.

15.4.1. Классы CSS

Простейший способ применения JavaScript для воздействия на стилизацию содержимого документа предусматривает добавление и удаление имен классов CSS из атрибута `class` в HTML-дескрипторах. Это легко делать с помощью свойства `classList` объектов `Element`, как объяснялось в подразделе "Атрибут `class`" ранее в главе.

Предположим, например, что таблица стилей вашего документа включает определение для класса `hidden`:

```
.hidden {
  display:none;
}
```

Определив такой стиль, вы можете скрывать (и затем показывать) элемент посредством кода следующего вида:

```
//Предположим, что этот элемент "tooltip" имеет class="hidden" в HTML-файле
// Вот как мы можем сделать его видимым:
document.querySelector("#tooltip").classList.remove("hidden");

// А так мы можем его скрыть:
document.querySelector("#tooltip").classList.add("hidden");
```

15.4.2. Встроенные стили

Чтобы продолжить предыдущий пример с всплывающей подсказкой, предположим, что документ структурирован только с одним элементом всплывающей подсказки, и мы хотим динамически позиционировать его перед отображением. В общем случае мы не можем создать в таблице стилей свой класс для каждого возможного местоположения всплывающей подсказки, так что свойство `classList` не поможет нам с позиционированием.

В такой ситуации нам необходимо настраивать атрибут `style` элемента всплывающей подсказки, устанавливая встроенные стили, которые специфичны для этого одного элемента. Модель DOM определяет свойство `style` для всех объектов `Element`, которые соответствуют атрибуту `style`. Однако в отличие от большинства таких свойств свойство `style` — не строка. Напротив, оно является объектом `CSSStyleDeclaration`: проанализированным представлением стилей CSS, которые в текстовой форме находятся в атрибуте `style`. Чтобы отобразить и установить местоположение нашей гипотетической всплывающей подсказки через JavaScript, мы можем использовать такой код:

```
function displayAt(tooltip, x, y) {
    tooltip.style.display = "block";
    tooltip.style.position = "absolute";
    tooltip.style.left = `${x}px`;
    tooltip.style.top = `${y}px`;
}
```

Соглашения об именовании: свойства CSS в JavaScript

Многие свойства стилей CSS, такие как `font-size`, содержат дефисы в своих именах. В JavaScript дефис интерпретируется как знак "минус" и применять его в именах свойств и других идентификаторах не разрешено. Следовательно, имена свойств объекта `CSSStyleDeclaration` слегка отличаются от имен фактических свойств CSS. Если имя свойства CSS содержит один или большее количество дефисов, то имя свойства `CSSStyleDeclaration` формируется путем удаления дефисов и преобразования буквы, находящейся непосредственно после каждого дефиса, в заглавную. Скажем, доступ к CSS-свойству `border-left-width` осуществляется через JavaScript-свойство `borderLeftWidth`, а CSS-свойство `font-family` записывается в коде JavaScript как `fontFamily`.

При работе со свойствами стиля в объекте `CSSStyleDeclaration` не забывайте о том, что все значения должны быть указаны в виде строк. В таблице стилей или атрибуте `style` вы можете записать:

```
display: block; font-family: sans-serif; background-color: #ffffff;
```

Чтобы сделать то же самое для элемента `e` в коде JavaScript, вам придется поместить все значения в кавычки:

```
e.style.display = "block";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
```

Обратите внимание на то, что точки с запятой находятся вне строк. Они представляют собой нормальные точки с запятой JavaScript; точки с запятой, используемые в таблицах стилей CSS, не требуются в качестве части строковых значений, которые вы устанавливаете в коде JavaScript.

Вдобавок не забывайте, что многие свойства CSS требуют указания единиц вроде "px" для пикселей или "pt" для пунктов. Таким образом, устанавливать свойство `marginLeft`, как показано ниже, некорректно:

```
e.style.marginLeft = 300;       // Некорректно: это число, а не строка.
e.style.marginLeft = "300";     // Некорректно: отсутствуют единицы.
```

При установке свойств стиля в коде JavaScript единицы обязательны, в точности как при установке свойств стиля в таблицах стилей. Вот как правильно установить значение свойства `marginLeft` элемента `e` в 300 пикселей:

```
e.style.marginLeft = "300px";
```

Если вы хотите установить свойство CSS в вычисленное значение, тогда обязательно добавьте единицы в конце вычисления:

```
e.style.left = `${x0 + left_border + left_padding}px`;
```

Вспомните, что некоторые свойства CSS, такие как `margin`, являются сокращениями для других свойств, подобных `margin-top`, `margin-right`, `margin-bottom` и `margin-left`. Объект `CSSStyleDeclaration` имеет свойства, которые соответствуют таким сокращенным свойствам. Например, вы можете установить свойство `margin` следующим образом:

```
e.style.margin = `${top}px ${right}px ${bottom}px ${left}px`;
```

Иногда вам может быть проще устанавливать или запрашивать встроенный стиль элемента как одиночное строковое значение, а не объект `CSSStyle Declaration`. Для этого вы можете применять методы `getAttribute()` и `setAttribute()` объекта `Element` или использовать свойство `cssText` объекта `CSSStyleDeclaration`:

```
// Копировать встроенные стили элемента e в элемент f:
f.setAttribute("style", e.getAttribute("style"));

// Или делать это следующим образом:
f.style.cssText = e.style.cssText;
```

При запрашивании свойства `style` элемента имейте в виду, что оно представляет только встроенные стили элемента, а большинство стилей для большинства элементов указывается в таблицах стилей вместо того, чтобы быть встроенными. К тому же в значениях, которые вы получаете при запрашивании свойства `style`, будут применяться любые единицы и любой формат сокращенного свойства, фактически используемые в HTML-атрибуте, а потому вашему коду, возможно, придется выполнять сложный анализ для их интерпретации. В общем случае, если вы хотите запрашивать стили элемента, то вам, вероятно, понадобится вычисляемый стиль, который обсуждается далее.

15.4.3. Вычисляемые стили

Вычисляемый стиль для элемента — это набор значений свойств, которые браузер выводит (или вычисляет) из встроенного стиля элемента, плюс все применимые правила стилей во всех таблицах стилей, т.е. набор свойств, действительно используемых для отображения элемента. Подобно встроенным стилям вычисляемые стили представляются с помощью объекта `CSSStyleDeclaration`. Тем не менее, в отличие от встроенных стилей вычисляемые стили допускают только чтение. Вы не можете устанавливать такие стили, но вычисляемый объект `CSSStyleDeclaration` для элемента позволяет выяснить, какие значения свойств стиля браузер применял при визуализации этого элемента.

Чтобы получить вычисляемый стиль для элемента, необходимо вызвать метод `getComputedStyle()` объекта `Window`. В первом аргументе методу передается элемент, вычисляемый стиль которого интересует. Необязательный второй аргумент используется для указания псевдоэлемента CSS, такого как `"::before"` или `"::after"`:

```
let title = document.querySelector("#section1title");
let styles = window.getComputedStyle(title);
let beforeStyles = window.getComputedStyle(title, "::before");
```

Возвращаемым значением `getComputedStyle()` является объект `CSSStyleDeclaration`, который представляет все стили, примененные к указанному элементу (псевдоэлементу). Между объектом `CSSStyleDeclaration`, представляющим встроенные стили, и объектом `CSSStyleDeclaration`, который представляет вычисляемые стили, существует несколько важных отличий.

- Свойства вычисляемых стилей предназначены только для чтения.
- Свойства вычисляемых стилей абсолютны: относительные единицы вроде процентных соотношений и пунктов преобразуются в абсолютные величины. Любое свойство, которое указывает размер (скажем, размер границы или размер шрифта), будет иметь значение, измеренное в пикселях. Такое значение будет строкой с суффиксом `px`, а потому вам по-прежнему придется его анализировать, но не нужно беспокоиться об анализе или преобразовании других единиц. Свойства, значения которых представляют цвета, будут возвращаться в формате `rgb()` или `rgba()`.
- Сокращенные свойства не вычисляются — вычисляются только фундаментальные свойства, на которых они основаны. Например, не запрашивайте свойство `margin`, но используйте `marginLeft`, `marginTop` и т.д. Аналогично не запрашивайте `border` или даже `borderWidth`. Взамен применяйте `borderLeftWidth`, `borderTopWidth` и т.д.
- Свойство `cssText` вычисляемого стиля является `undefined`.

Объект `CSSStyleDeclaration`, возвращаемый `getComputedStyle()`, обычно содержит гораздо больше информации об элементе, чем объект `CSSStyleDeclaration`, полученный из свойства встроенного стиля данного элемента. Но вычисляемые стили могут быть сложными, и их запрашивание

не всегда предоставляет информацию, которую вы могли ожидать. Возьмем атрибут `font-family`: он принимает список разделенных запятыми желательных семейств шрифтов для обеспечения межплатформенной переносимости. Когда вы запрашиваете свойство `fontFamily` вычисляемого стиля, то просто получаете значение наиболее специфичного стиля `font-family`, который применяется к элементу. Может возвратиться значение наподобие `"arial,helvetica,sans-serif"`, которое ничего не говорит о том, какая гарнитура шрифта в действительности используется. Аналогично, если элемент не позиционируется абсолютным образом, то попытка запросить его позицию и размер через свойства `top` и `left` через вычисляемый стиль элемента часто приводит к возвращению значения `auto`. Это совершенно законное значение CSS, но вероятно не то, что вы искали.

Хотя CSS можно применять для точного указания позиции и размера элементов документа, запрашивание вычисляемого стиля какого-то элемента не является предпочтительным способом определения размера и позиции элемента. Более простая и переносимая альтернатива описана в подразделе 15.5.2.

15.4.4. Работа с таблицами стилей в сценариях

Помимо атрибутов классов и встроенных стилей в коде JavaScript можно также манипулировать самими таблицами стилей. Таблицы стилей ассоциируются с HTML-документом посредством дескриптора `<style>` или `<link rel="stylesheet">`. Оба они — обыкновенные HTML-дескрипторы, поэтому вы можете снабдить их атрибутами `id` и затем находить с помощью `document.querySelector()`.

Объекты `Element` для дескрипторов `<style>` и `<link>` имеют свойство `disabled`, которое вы можете использовать для отключения целой таблицы стилей. Ниже приведен пример работы со свойством `disabled`:

```
// Эта функция осуществляет переключение между "светлой" и "темной" темами
function toggleTheme() {
    let lightTheme = document.querySelector("#light-theme");
    let darkTheme = document.querySelector("#dark-theme");
    if (darkTheme.disabled) {     // В текущий момент светлая тема,
                                  // переключить на темную.
        lightTheme.disabled = true;
        darkTheme.disabled = false;
    } else {                      // В текущий момент темная тема,
                                  // переключить на светлую.
        lightTheme.disabled = false;
        darkTheme.disabled = true;
    }
}
```

Еще один простой способ работы с таблицами стилей в коде — вставка новых таблиц стилей в документ с применением методик манипулирования моделью DOM, которые вы уже видели.

Вот пример:

```
function setTheme(name) {
    // Создать новый элемент <link rel="stylesheet">
    // для загрузки именованной таблицы стилей.
    let link = document.createElement("link");
    link.id = "theme";
    link.rel = "stylesheet";
    link.href = `themes/${name}.css`;

    // Искать существующую ссылку с идентификатором "theme".
    let currentTheme = document.querySelector("#theme");
    if (currentTheme) {
        // Если тема существует, то заменить ее новой темой.
        currentTheme.replaceWith(link);
    } else {
        // В противном случае просто вставить
        // ссылку на таблицу стилей темы.
    document.head.append(link);
    }
}
```

Менее явно вы также можете вставить в свой документ строку HTML-разметки, содержащую дескриптор <style>. Это забавный трюк, например:

```
document.head.insertAdjacentHTML(
    "beforeend",
    "<style>body{transform:rotate(180deg)}</style>"
);
```

Браузеры определяют API-интерфейс, который позволяет коду JavaScript заглядывать внутрь таблиц стилей для запрашивания, модифицирования, вставки и удаления в них правил стилей. Этот API-интерфейс настолько специализирован, что здесь не документируется. Вы можете почитать о нем в MDN, выполнив поиск по ключевым словам "CSSStyleSheet" и "CSS Object Model".

15.4.5. Анимация и события CSS

Предположим, что у вас есть следующие два класса CSS, определенные в таблице стилей:

```
.transparent { opacity: 0; }
.fadeable { transition: opacity .5s ease-in }
```

Если вы примените к элементу первый стиль, тогда он станет полностью прозрачным и оттого невидимым. Но если вы примените второй стиль, который сообщает браузеру о том, что когда непрозрачность элемента изменяется, такое изменение должно быть подвергнуто анимации на протяжении 0,5 секунды, а `ease-in` указывает, что анимация изменения непрозрачности должна начинаться медленно и затем ускориться.

Теперь предположим, что ваш HTML-документ содержит элемент с классом `"fadeable"`:

```
<div id="subscribe" class="fadeable notification">...</div>
```

В коде JavaScript вы можете добавить класс "transparent":

```
document.querySelector("#subscribe").classList.add("transparent");
```

Этот элемент сконфигурирован для выполнения анимации изменений непрозрачности. Добавление класса "transparent" изменяет непрозрачность и запускает анимацию: браузер обеспечивает "постепенное исчезновение" элемента, так что в течение 0,5 секунды он становится полностью прозрачным.

Подход работает и в обратном направлении: если вы удалите класс "transparent" из "постепенно исчезающего" элемента, то непрозрачность также изменится, и элемент постепенно появится и снова станет видимым.

Код JavaScript не обязан делать какую-то работу, чтобы такая анимация произошла: она является чистым эффектом CSS. Но код JavaScript можно использовать для ее запуска.

Код JavaScript можно применять также для отслеживания продвижения перехода CSS, поскольку веб-браузер инициирует события в начале и конце перехода. Событие "transitionrun" посылается при первом запуске перехода. Это может произойти до начала любых визуальных изменений, когда был указан стиль transition-delay. Как только начинаются визуальные изменения, посылается событие "transitionstart", а когда анимация завершается, посылается событие "transitionend". Целью всех упомянутых событий является, конечно же, подвергающийся анимации элемент. Объектом события, передаваемым обработчикам, будет TransitionEvent. Он имеет свойство propertyName, которое указывает анимированное свойство CSS, и свойство elapsedTime, которое для событий "transitionend" указывает, сколько секунд прошло с момента возникновения события "transitionstart".

В дополнение к переходам CSS также поддерживает более сложную форму анимации, известную просто как "анимация CSS". Для определения деталей анимации она использует свойства CSS, такие как animation-name и animation-duration, и специальное правило @keyframes. Детали работы анимации CSS выходят за рамки настоящей книги, но стоит повторить: если вы определяете все свойства анимации для класса CSS, тогда можете применять JavaScript для запуска анимации, добавив класс к элементу, который должен быть подвергнут анимации.

И подобно переходам CSS анимация CSS также инициирует события, которые ваш код JavaScript может прослушивать. Событие "animationstart" посылается, когда анимация стартует, а событие "animationend" — когда она завершается. Если анимация повторяется более одного раза, то после каждого повторения кроме последнего посылается событие "animationiteration". Целью события является анимированный элемент, а передаваемым функциям обработчиков объектом события — объект AnimationEvent. Такие события включают свойство animationName, которое задает свойство animation-name, определяющее анимацию, и свойство elapsedTime, указывающее количество прошедших с начала анимации секунд.

15.5. Геометрия и прокрутка документов

До сих пор в главе мы рассматривали документы как абстрактные деревья элементов и текстовых узлов. Но когда браузер визуализирует документ внутри окна, он создает визуальное представление документа, в котором каждый элемент имеет позицию и размер. Веб-приложения часто могут трактовать документы как деревья элементов и никогда не заботиться о том, каким образом эти элементы визуализируются на экране. Однако временами необходимо определять точную геометрию элемента. Скажем, если вы хотите использовать CSS для динамического позиционирования элемента (такого как всплывающая подсказка) рядом с каким-то обыкновенным элементом, который позиционирован браузером, тогда вам нужна возможность определять местоположение данного элемента.

В последующих подразделах объясняется, как перемещаться между абстрактной, древовидной моделью документа и геометрическим, основанным на координатах представлением документа в том виде, в каком он размещен в окне браузера.

15.5.1. Координаты документа и координаты окна просмотра

Позиция элемента документа измеряется в пикселях CSS, причем координата x увеличивается при движении вправо, а координата y увеличивается по мере движения вниз. Тем не менее, есть две разные точки, которые допускается применять в качестве начала системы координат: координаты x и y элемента могут быть относительными левого верхнего угла документа или левого верхнего угла *окна просмотра* (viewport), где документ отображается. В окнах и вкладках верхнего уровня "окно просмотра" — это часть браузера, которая фактически отображает содержимое документа: в нее не входят "украшения" браузера, такие как меню, панели инструментов и вкладки. Что касается документов, отображаемых в дескрипторах `<iframe>`, то окно просмотра для вложенного документа определяется элементом внутреннего фрейма в DOM. В любом случае, когда мы говорим о позиции элемента, то должны четко понимать, какие координаты используются — документа или окна просмотра. (Имейте в виду, что координаты окна просмотра иногда называют "оконными координатами".)

Если документ меньше, чем окно просмотра, или если он не прокручивался, тогда левый верхний угол документа является левым верхним углом окна просмотра и системы координат документа и окна просмотра совпадают. Однако в общем случае для преобразования между двумя системами координат мы должны добавлять или вычитать смещения прокрутки. Скажем, если элемент имеет координату y, равную 200 пикселей в координатах документа, а пользователь выполнил прокрутку вниз на 75 пикселей, тогда элемент имеет координату y, равную 125 пикселей в координатах окна просмотра. Аналогично, если элемент имеет координату x, равную 400 пикселей в координатах окна просмотра после того, как пользователь выполнил прокрутку окна просмотра на 200 пикселей по горизонтали, тогда координата x элемента в координатах документа будет составлять 600 пикселей.

Если мы обратимся к умозрительной модели документов, напечатанных на бумаге, то логично предположить, что каждый элемент в документе обязан иметь уникальную позицию в координатах документа вне зависимости от того, насколько пользователь прокручивал документ. Это привлекательное качество бумажных документов и оно применимо к простым веб-документам, но в действительности координаты документа в веб-сети не работают. Проблема в том, что CSS-свойство `overflow` разрешает элементам внутри документа включать в себя больше содержимого, чем они в состоянии отобразить. Элементы могут иметь собственные линейки прокрутки и служить окнами просмотра для находящегося в них содержимого. Тот факт, что веб-сеть допускает наличие прокручиваемых элементов внутри прокручиваемого документа, означает невозможность описать позицию элемента внутри документа с использованием одиночной точки (x,y).

Поскольку координаты документа в действительности не работают, в коде JavaScript стороны клиента обычно применяются координаты окна просмотра. Например, описываемые далее методы `getBoundingClientRect()` и `elementFromPoint()` используют координаты окна просмотра и в данной системе координат представлены также свойства `clientX` и `clientY` объектов событий мыши и указателя.

Когда вы явно позиционируете элемент с применением CSS-свойства `position:fixed`, свойства `top` и `left` интерпретируются в координатах окна просмотра. В случае использования `position:relative` элемент позиционируется относительно того места, где он располагался бы, если бы для него не устанавливалось свойство `position`. Если вы применяете `position:absolute`, тогда свойства `top` и `left` будут браться относительно документа или ближайшего вмещающего позиционированного элемента. Это означает, например, что абсолютно позиционированный элемент внутри относительно позиционированного элемента позиционируется относительно контейнерного элемента, а не всего документа. Временными очень удобно создавать относительно позиционированный контейнер со свойствами `top` и `left`, равными 0 (так что контейнер нормально компонуется), чтобы установить новое начало системы координат для абсолютно позиционированных элементов, которые он содержит. Мы могли бы назвать такую новую систему координат "координатами контейнера", чтобы отличать ее от координат документа и координат окна просмотра.

Пиксели CSS

Если вы, как и я, достаточно взрослые, что помните компьютерные мониторы с разрешающей способностью 1024 × 768 и телефоны с сенсорными экранами с разрешающей способностью 320 × 480, то все еще можете полагать, что слово "пиксель" относится к одиночному "элементу изображения" в оборудовании. Современные мониторы 4K и дисплеи Retina обладают настолько высоким разрешением, что программные пиксели были отделены от аппаратных пикселей. Пиксель CSS — и, следовательно, пиксель JavaScript стороны клиента — на самом деле может состоять из множества пиксе-

лей устройства. Свойство devicePixelRatio объекта Window указывает, сколько пикселей устройства используется для каждого программного пикселя. Скажем, соотношение пикселей устройства, равное 2, говорит о том, что каждый программный пиксель в действительности является сеткой 2×2 из аппаратных пикселей. Значение devicePixelRatio зависит от физической разрешающей способности оборудования, от настроек в вашей операционной системе и от степени масштабирования в вашем браузере.

Значение devicePixelRatio не обязано быть целым числом. Если CSS-свойство font-size установлено в 12px и соотношение пикселей устройства составляет 2.5, тогда фактическим размером шрифта в пикселях устройства будет 30. Из-за того, что значения в пикселях, применяемые в CSS, больше не соответствуют напрямую индивидуальным пикселям на экране, координаты в пикселях больше не должны быть целыми числами. Если devicePixelRatio равно 3, тогда координата 3.33 вполне осмысленна. А если devicePixelRatio равно 2, то координата 3.33 будет просто округляться с повышением до 3.5.

15.5.2. Запрашивание геометрии элемента

Вы можете определить размер (включая границу и отступы CSS, но не поля) и позицию (в координатах окна просмотра) элемента, вызывая его метод getBoundingClientRect(). Метод не принимает аргументов и возвращает объект со свойствами left, right, top, bottom, width и height. Свойства left и top дают координаты x и y левого верхнего угла элемента, а свойства right и bottom — координаты правого нижнего угла. Разности между этими значениями будут свойствами width и height.

Блочные элементы, такие как изображения, абзацы и элементы , всегда прямоугольны, когда компонуются браузером. Тем не менее, встроенные элементы вроде , <code> и <b> могут распространяться на несколько строк и потому состоять из множества прямоугольников. Представьте себе, например, что определенный текст внутри дескрипторов и отображается с переносом на вторую строку. Его прямоугольники состоят из конца первой строки и начала второй строки. Если вы вызовите getBoundingClientRect() на таком элементе, то ограничивающий прямоугольник будет включать полную ширину обеих строк. Если вы хотите запросить индивидуальные прямоугольники встроенных элементов, тогда вызовите метод getClientRects(), чтобы получить допускающий только чтение объект, похожий на массив, элементами которого являются объекты прямоугольников, подобные тем, что возвращает метод getBoundingClientRect().

15.5.3. Определение элемента в точке

Метод getBoundingClientRect() позволяет определять текущую позицию элемента в окне просмотра. Иногда необходимо двигаться в другом направлении и определять, какой элемент находится в заданном местоположении внутри окна

просмотра. Это можно делать с помощью метода elementFromPoint() объекта Document. Вызывайте его с координатами x и y точки (используя координаты окна просмотра, а не координаты документа: скажем, подойдут координаты clientX и clientY события мыши). Метод elementFromPoint() возвращает объект Element, который находится в указанной позиции. Алгоритм обнаружения попаданий для выбора элемента точно не определен, но замысел метода elementFromPoint() заключается в том, чтобы возвратить самый внутренний (наиболее глубоко вложенный) и самый верхний (с наивысшим CSS-атрибутом z-index) элемент в данной точке.

15.5.4. Прокрутка

Метод scrollTo() объекта Window принимает координаты x и y точки (в координатах документа) и устанавливает их как смещения линейки прокрутки. То есть он прокручивает окно, чтобы указанная точка оказалась в левом верхнем узлу окна просмотра. Если вы зададите точку, которая расположена слишком близко к низу или слишком близко к правому краю документа, тогда браузер переместит ее как можно ближе к левому верхнему углу, но не будет в состоянии пройти весь путь до конца. Следующий код заставляет браузер выполнить прокрутку так, чтобы стала видимой самая нижняя страница документа:

```
// Получить высоты документа и окна просмотра.
let documentHeight = document.documentElement.offsetHeight;
let viewportHeight = window.innerHeight;

// Выполнить прокрутку, чтобы в окне просмотра
// появилась последняя "страница".
window.scrollTo(0, documentHeight - viewportHeight);
```

Метод scrollBy() объекта Window похож на scrollTo(), но его аргументы относительны и добавляются к текущей позиции прокрутки:

```
// Выполнять прокрутку на 50 пикселей вниз каждые 500 мс.
// Обратите внимание, что нет никакого способа отключить это!
setInterval(() => { scrollBy(0,50)}, 500);
```

Если вы хотите плавно прокручивать с помощью метода scrollTo() или scrollBy(), тогда передавайте взамен двух чисел одиночный объектный аргумент:

```
window.scrollTo({
    left: 0,
    top: documentHeight - viewportHeight,
    behavior: "smooth"
});
```

Часто вместо прокрутки до числового местоположения в документе нам нужно всего лишь прокрутить, чтобы определенный элемент в документе стал видимым. Для этого можно вызвать метод scrollIntoView() на желаемом HTML-элементе. Метод scrollIntoView() гарантирует, что элемент, на котором он вызван, будет виден в окне просмотра. По умолчанию он пытается поместить верхний край элемента поблизости к верху окна просмотра. Если

в качестве единственного аргумента передается false, тогда метод пытается поместить нижний край элемента в низ окна просмотра. При необходимости браузер будет прокручивать окно просмотра и по горизонтали, чтобы сделать элемент видимым.

Вы также можете передавать методу scrollIntoView() объект с установленным свойством behavior:"smooth" для обеспечения плавной прокрутки. Вы можете устанавливать свойство block для указания, где элемент должен позиционироваться по вертикали, и свойство inline, чтобы указать, как он должен позиционироваться по горизонтали, если требуется горизонтальная прокрутка. Допустимые значения для обоих свойств — start, end, nearest и center.

15.5.5. Размер окна просмотра, размер содержимого и позиция прокрутки

Как уже обсуждалось, окна браузера и остальные HTML-элементы способны отображать прокручиваемое содержимое. В такой ситуации временами нам нужно знать размер окна просмотра, размер содержимого и смещения прокрутки содержимого внутри окна просмотра. В этом разделе раскрываются все детали.

Для окон браузера размер окна просмотра выдается свойствами window.innerWidth и window.innerHeight. (Веб-страницы, оптимизированные для мобильных устройств, часто применяют дескриптор <meta name="viewport"> в своих дескрипторах <head>, чтобы устанавливать желаемую ширину окна просмотра для страницы.) Общий размер документа будет таким же, как размер элемента <html>, т.е. document.documentElement. Чтобы получить ширину и высоту документа, вы можете вызывать метод getBoundingClientRect() объекта document.documentElement или воспользоваться свойствами offsetWidth и offsetHeight объекта document.documentElement. Смещения прокрутки документа внутри окна просмотра доступны как window.scrollX и window.scrollY. Они являются свойствами, допускающими только чтение, поэтому их нельзя устанавливать с целью прокрутки документа: взамен применяйте метод window.scrollTo().

Что касается элементов, то ситуация чуть сложнее. В каждом объекте Element определены следующие три группы свойств:

```
offsetWidth     clientWidth     scrollWidth
offsetHeight    clientHeight    scrollHeight
offsetLeft      clientLeft      scrollLeft
offsetTop       clientTop       scrollTop
offsetParent
```

Свойства offsetWidth и offsetHeight элемента возвращают его экранный размер в пикселях CSS. Возвращаемые размеры включают границу и отступы элемента, но не поля. Свойства offsetLeft и offsetTop возвращают координаты x и y элемента. Для многих элементов их значения являются координатами документа. Но для потомков позиционированных элементов и для ряда других

элементов, таких как ячейки таблицы, эти свойства возвращают координаты относительно элемента предка, а не самого документа. Свойство `offsetParent` указывает, относительно какого элемента свойства заданы. Все свойства смещения допускают только чтение.

Свойства `clientWidth` и `clientHeight` подобны свойствам `offsetWidth` и `offsetHeight`, но не включают размер границы — только область содержимого и отступы. Свойства `clientLeft` и `clientTop` не особенно полезны: они возвращают расстояние по горизонтали и по вертикали между внешней стороной отступа элемента и внешней стороной его границы. Обычно такие значения представляют собой просто ширину левой и верхней границы. Все свойства клиента допускают только чтение. Для встроенных элементов вроде `<i>`, `<code>` и `<span>` все они возвращают 0.

Свойства `scrollWidth` и `scrollHeight` возвращают размер области содержимого элемента плюс его отступы плюс выходящее за пределы содержимое. Когда содержимое умещается внутри области содержимого, не выходя за ее пределы, эти свойства будут иметь такие же значения, как у `clientWidth` и `clientHeight`. Но когда имеется выход за пределы, они включают выходящее за пределы содержимое и возвращают значения, превышающие `clientWidth` и `clientHeight`. Свойства `scrollLeft` и `scrollTop` дают смещение прокрутки содержимого элемента внутри окна просмотра элемента. В отличие от всех остальных описанных здесь свойств `scrollLeft` и `scrollTop` — записываемые свойства, которые вы можете устанавливать для выполнения прокрутки содержимого внутри элемента. (В большинстве браузеров объекты `Element` также имеют методы `scrollTo()` и `scrollBy()`, как у объекта `Window`, но они пока не поддерживаются повсеместно.)

15.6. Веб-компоненты

HTML является языком для разметки документов и определяет широкий набор дескрипторов для этой цели. За последние три десятилетия он стал языком, который используется для описания пользовательских интерфейсов веб-приложений, но базовые HTML-дескрипторы вроде `<input>` и `<button>` не отвечают требованиям современного дизайна пользовательского интерфейса. Разработчики веб-приложений способны заставить его работать, но только за счет применения CSS и JavaScript с целью дополнения внешнего вида и поведения базовых HTML-дескрипторов. Рассмотрим типичный компонент пользовательского интерфейса, такой как поле поиска, показанное на рис. 15.3.

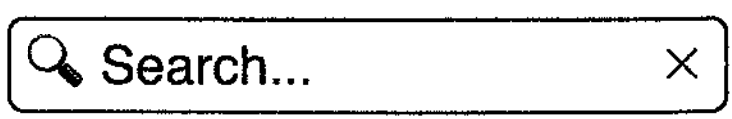


Рис. 15.3. Компонент пользовательского интерфейса — поле поиска

HTML-элемент <input> может использоваться для приема одиночной строки ввода от пользователя, но у него нет возможности отображать значки, подобные увеличительному стеклу слева и значку отмены X справа. Чтобы реализовать элемент современного пользовательского интерфейса такого рода для веб-сети, нам необходимо задействовать, по меньшей мере, четыре HTML-элемента: элемент <input> для приема и отображения ввода пользователя, два элемента <img> (или в данном случае два элемента , отображающие глифы Unicode) и контейнерный элемент для удержания перечисленных трех дочерних элементов. Кроме того, нам придется применить CSS, чтобы скрыть стандартную границу элемента <input>, и определить границу для контейнера. И нам необходимо использовать JavaScript, чтобы заставить все HTML-элементы работать вместе. Когда пользователь щелкает на значке X, нам нужен обработчик событий для очистки ввода в элементе <input>, например.

Такой крупный объем работ нужно выполнять каждый раз, когда желательно отобразить поле поиска в веб-приложении, а потому большинство веб-приложений в наши дни не пишутся с применением "сырого" HTML. Взамен многие разработчики веб-приложений используют фреймворки вроде React и Angular, которые поддерживают создание многократно применяемых компонентов пользовательского интерфейса, подобных показанному здесь полю поиска. Веб-компоненты — это встроенная в браузер альтернатива таким фреймворкам, основанная на трех относительно недавних добавлениях к веб-стандартам, которые позволяют JavaScript расширять HTML новыми дескрипторами, работающими как самостоятельные многократно используемые компоненты пользовательского интерфейса.

В последующих подразделах объясняется, как применять внутри своих веб-страниц веб-компоненты, определяемые другими разработчиками, затем обсуждается каждая из трех технологий, на которых основаны веб-компоненты, и в заключение все три технологии увязываются вместе в рамках примера, где реализуется элемент типа поля поиска, представленный на рис. 15.3.

15.6.1. Использование веб-компонентов

Веб-компоненты определяются в JavaScript, поэтому для использования веб-компонента в своем HTML-файле понадобится включить JavaScript-файл, в котором компонент определен. Поскольку веб-компоненты — относительно новая технология, они часто записываются в виде модулей JavaScript, так что вот как вы могли бы включить подобный модуль в свой HTML-файл:

```
<script type="module" src="components/search-box.js">
```

Веб-компоненты определяют собственные имена HTML-дескрипторов с важным ограничением, касающимся того, что имена их дескрипторов обязаны включать дефис. (Это означает, что в будущих версиях HTML могут быть введены новые дескрипторы без дефисов, а потому нет шанса возникновения конфликта с любым веб-компонентом.) Для применения веб-компонента просто используйте его дескриптор в своем HTML-файле:

```
<search-box placeholder="Search..."></search-box>
```

Веб-компоненты могут иметь атрибуты в точности как обыкновенные HTML-дескрипторы; в документации по используемому компоненту должно сообщаться, какие атрибуты поддерживаются. Веб-компоненты не могут определяться с помощью самозакрывающихся дескрипторов. Скажем, нельзя записывать `<search-box/>`. Ваш HTML-файл обязан включать и открывающий, и закрывающий дескрипторы. Подобно обыкновенным HTML-элементам одни веб-компоненты написаны так, что они рассчитывают на дочерние элементы, а другие так, что дочерних элементов они не ожидают (и не будут их отображать). Некоторые веб-компоненты написаны так, что они могут дополнительно принимать специально помеченные дочерние элементы, которые будут появляться в именованных "слотах". Компонент `<search-box>`, показанный на рис. 15.3 и реализованный в примере 15.3, применяет "слоты" для двух значков, которые он отображает. Если вы хотите использовать `<search-box>` с другими значками, тогда можете применить такую HTML-разметку:

```
<search-box>
  <img src="images/search-icon.png" slot="left"/>
  <img src="images/cancel-icon.png" slot="right"/>
</search-box>
```

Атрибут `slot` является расширением HTML, которое используется для указания на то, какие дочерние элементы и куда должны поступать. Имена слотов — в этом примере `"left"` и `"right"` — определены веб-компонентом. Если применяемый вами компонент поддерживает слоты, то данный факт должен быть отражен в документации.

Ранее я отмечал, что веб-компоненты часто реализуются в виде модулей JavaScript и могут загружаться в HTML-файлы с помощью дескриптора `<script type="module">`. Вы можете вспомнить из начала главы, что модули загружаются после синтаксического анализа содержимого документа, как если бы они имели атрибут `defer`. Таким образом, это означает, что веб-браузер обычно будет анализировать и визуализировать дескрипторы вроде `<search-box>` до того, как запустит код, который сообщит ему, чем является `<search-box>`. При использовании веб-компонентов это нормально. Синтаксические анализаторы HTML в веб-браузерах являются гибкими и очень снисходительными к входным данным, которые они не понимают. Когда они встречают дескриптор веб-компонента до того, как компонент был определен, то добавляют в дерево DOM обобщенный элемент `HTMLElement`, хотя и не знают, что с ним делать. Позже, когда специальный элемент определился, обобщенный элемент "модернизируется", чтобы выглядеть и вести себя желательным образом.

Если веб-компонент имеет дочерние элементы, то до определения компонента они, вероятно, будут отображаться некорректно. Вы можете применить следующий стиль CSS для удержания веб-компонентов скрытыми до тех вор, пока они не будут определены:

```
/*
 * Сделать компонент <search-box> невидимым до его определения.
 * Попытайтесь продублировать его конечную компоновку и размер, чтобы
 * соседнее содержимое не перемещалось, когда он станет определенным.
 */
```

```
search-box:not(:defined) {
   opacity:0;
   display: inline-block;
   width: 300px;
   height: 50px;
}
```

Подобно обыкновенным HTML-элементам веб-компоненты можно использовать в JavaScript. Если вы включили дескриптор <search-box> в свою веб-страницу, тогда можете посредством метода `querySelector()` и соответствующего селектора CSS получить ссылку на него, как поступали бы в отношении любого другого HTML-дескриптора. Обычно предпринимать такое действие имеет смысл только после того, как определяющий компонент модуль выполнился, так что будьте осторожны, чтобы не запросить веб-компоненты, созданные не вами, слишком рано. Реализации веб-компонентов, как правило (но это не требование), определяют свойство JavaScript для каждого атрибута HTML, который они поддерживают. И подобно HTML-элементам они могут также определять полезные методы. В документации по применяемому веб-компоненту должно быть указано, какие свойства и методы доступны в коде JavaScript.

Теперь, когда вы знаете, как использовать веб-компоненты, в последующих трех подразделах рассматриваются три функциональных возможности веб-браузеров, которые позволяют их реализовать.

Узлы `DocumentFragment`

Прежде чем мы сможем раскрыть API-интерфейсы веб-компонентов, нам необходимо ненадолго возвратиться к API-интерфейсу DOM с целью объяснения, что такое `DocumentFragment`. API-интерфейс DOM организует документ в виде дерева объектов `Node`, где `Node` может быть `Document`, `Element`, узел `Text` или даже `Comment`. Ни один из этих типов не позволяет представлять фрагмент документа, который состоит из набора родственных узлов без их родителя. Именно здесь на помощь приходит `DocumentFragment` — еще один тип `Node`, который служит в качестве временного родителя, когда вы хотите манипулировать группой родственных узлов как единым целым. Вы можете создать узел `DocumentFragment`, вызвав `document.createDocumentFragment()`. Результирующий объект `DocumentFragment` вы можете применять подобно объекту `Element` и посредством `append()` добавлять к нему содержимое. Объект `DocumentFragment` отличается от `Element`, потому что он не имеет родителя. Но более важно то, что при вставке узла `DocumentFragment` в документ сам `DocumentFragment` не вставляется. Взамен вставляются все его дочерние элементы.

15.6.2. Шаблоны HTML

HTML-дескриптор <template> имеет лишь слабое отношение к веб-компонентам, но он делает возможной полезную оптимизацию для компонентов, которые часто встречаются на веб-страницах. Дескрипторы <template> и их дочерние элементы никогда не визуализируются веб-браузером и приносят пользу

только на веб-страницах, которые используют JavaScript. Идея, положенная в основу этого дескриптора, заключается в том, что когда веб-страница содержит много повторений той же самой базовой HTML-структуры (такой как строки в таблицы или внутренняя реализация веб-компонента), мы можем применить `<template>` для определения такой структуры один раз, после чего использовать JavaScript для ее дублирования столько раз, сколько нужно.

В JavaScript дескриптор `<template>` представлен объектом `HTMLTemplateElement`, определяющим единственное свойство `content`, значением которого является объект `DocumentFragment` со всеми дочерними узлами `<template>`. Вы можете клонировать `DocumentFragment` и затем по мере необходимости вставлять клонированный экземпляр в свой документ. Сам фрагмент вставляться не будет, но его дочерние узлы будут. Предположим, что вы работаете с документом, который включает дескрипторы `<table>` и `<template id="row">`, а шаблон определяет структуру строк для данной таблицы. Вы могли бы применять шаблон следующим образом:

```
let tableBody = document.querySelector("tbody");
let template = document.querySelector("#row");
let clone = template.content.cloneNode(true); //Глубокое клонирование
//...Использовать DOM для вставки содержимого в элементы <td> клона...
// Добавить клонированную и инициализированную строку в таблицу.
tableBody.append(clone);
```

Чтобы быть полезными, элементы шаблона не обязаны находиться буквально в HTML-документе. Вы можете создать шаблон в коде JavaScript, создать его дочерние элементы с помощью свойства `innerHTML` и далее создавать любое нужное количество клонов без накладных расходов на синтаксический анализ `innerHTML`. Именно так шаблоны HTML обычно используются в веб-компонентах, и описанная методика демонстрируется в примере 15.3.

15.6.3. Специальные элементы

Вторым функциональным средством веб-браузеров, которое делает возможными веб-компоненты, являются "специальные элементы": возможность ассоциировать класс JavaScript с именем дескриптора HTML, чтобы любые дескрипторы такого рода в документе автоматически превращались в экземпляры класса в дереве DOM. Метод `customElements.define()` принимает имя дескриптора веб-компонента в своем первом аргументе (не забывайте, что имя дескриптора обязано включать дефис) и подкласс класса `HTMLElement` — во втором. Любые элементы с таким именем дескриптора, существующие в документе, "модернизируются" до заново созданных экземпляров класса. При дальнейшем выполнении браузером синтаксического анализа HTML-разметки он будет автоматически создавать экземпляр класса для каждого встреченного дескриптора.

Передаваемый методу `customElements.define()` класс должен расширять класс `HTMLElement`, а не более специфичный тип вроде `HTMLButtonElement`[4].

[4] Спецификация специальных элементов разрешает создавать подклассы `<button>` и других классов специфичных элементов, но это не поддерживается в Safari, а для использования специального элемента, который расширяет что угодно кроме `HTMLElement`, требуется другой синтаксис.

Вспомните из главы 9, что когда класс JavaScript расширяет другой класс, функция конструктора обязана вызывать `super()` до применения ключевого слова `this`, поэтому если класс специального элемента имеет конструктор, то он должен вызвать `super()` (без аргументов), прежде чем делать что-нибудь другое.

Браузер будет автоматически вызывать определенные "методы жизненного цикла" класса специального элемента. Метод `connectedCallback()` вызывается при вставке специального элемента в документ, и многие элементы используют этот метод для выполнения инициализации. Также есть метод `disconnectedCallback()`, который вызывается, когда (и если) элемент удаляется из документа, хотя он применяется реже.

Если в классе специального элемента определено статическое свойство `observedAttributes`, значение которого представляет собой массив имен атрибутов, и если любой из именованных атрибутов устанавливается (или изменяется) в экземпляре специального элемента, тогда браузер вызовет метод `attributeChangedCallback()`, передавая ему имя атрибута, его старое значение и новое значение. Такой обратный вызов может предпринимать произвольные шаги, необходимые для обновления компонента на основе значений его атрибутов.

В классах специальных элементов могут также определяться любые другие свойства и методы, какие вы захотите. Как правило, в них будут определены методы получения и установки, которые делают атрибуты элемента доступными в виде свойств JavaScript.

Давайте рассмотрим пример специального элемента. Предположим, что нас интересует возможность отображения кругов внутри абзацев обыкновенного текста. Мы хотели бы писать HTML-разметку наподобие приведенной далее, чтобы визуализировать математические задачи, такие как показанная на рис. 15.4:

```
<p>
  The document has one marble: <inline-circle></inline-circle>.
  The HTML parser instantiates two more marbles:
  <inline-circle diameter="1.2em" color="blue"></inline-circle>
  <inline-circle diameter=".6em" color="gold"></inline-circle>.
  How many marbles does the document contain now?
</p>
```

Мы можем реализовать этот специальный элемент `<inline-circle>` с помощью кода, представленного в примере 15.2.

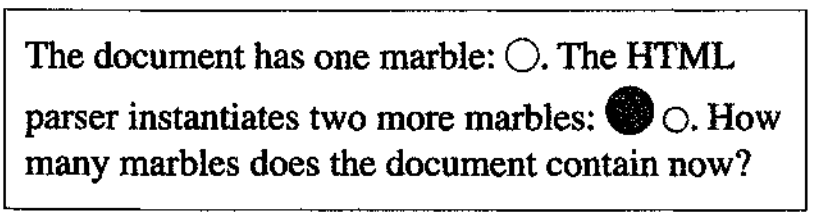


Рис. 15.4. Специальный элемент встраиваемого круга

Пример 15.2. Специальный элемент <inline-circle>

```
customElements.define("inline-circle",class InlineCircle extends HTMLElement {
    // Браузер вызывает этот метод, когда элемент <inline-circle>
    // вставляется в документ. Существует также метод
    // disconnectedCallback(), который в настоящем примере не нужен.
    connectedCallback() {
        // Установить стили, необходимые для создания кругов.
        this.style.display = "inline-block";
        this.style.borderRadius = "50%";
        this.style.border = "solid black 1px";
        this.style.transform = "translateY(10%)";

        // Если размер еще не определен, тогда установить стандартный
        // размер, который основан на текущем размере шрифта.
        if (!this.style.width) {
            this.style.width = "0.8em";
            this.style.height = "0.8em";
        }
    }

    // Статическое свойство observedAttributes указывает, об изменениях
    // каких атрибутов мы хотим получать уведомления.
    // (Мы используем здесь метод получения, поскольку static можно
    // применять только с методами.)
    static get observedAttributes() { return ["diameter", "color"]; }

    // Этот обратный вызов вызывается в случае изменения одного из
    // перечисленных выше атрибутов либо при синтаксическом анализе
    // специального элемента в первый раз, либо позже.
    attributeChangedCallback(name, oldValue, newValue) {
        switch(name) {
        case "diameter":
            // Если изменился атрибут diameter, тогда обновить стили размеров.
            this.style.width = newValue;
            this.style.height = newValue;
            break;
        case "color":
            // Если изменился атрибут color, тогда обновить стиль цвета.
            this.style.backgroundColor = newValue;
            break;
        }
    }

    // Определить свойства JavaScript, которые соответствуют атрибутам
    // элемента. Эти методы получения и установки просто получают
    // и устанавливают лежащие в основе атрибуты.
    // Установка свойства JavaScript приводит к установке атрибута, что
    // инициирует вызов метода attributeChangedCallback(), который
    // обновляет стили элемента.
    get diameter() { return this.getAttribute("diameter"); }
    set diameter(diameter) { this.setAttribute("diameter", diameter); }
    get color() { return this.getAttribute("color"); }
    set color(color) { this.setAttribute("color", color); }
    });
```

15.6.4. Теневая модель DOM

Специальный элемент, продемонстрированный в примере 15.2, плохо инкапсулирован. Когда вы устанавливаете его атрибуты `diameter` или `color`, он реагирует изменением собственного атрибута `style`, а такого поведения мы не можем ожидать от настоящего HTML-элемента. Чтобы превратить специальный элемент в подлинный веб-компонент, должен использоваться мощный механизм инкапсуляции, известный как *теневая модель DOM* (shadow DOM).

Теневая модель DOM позволяет прикреплять "корневой элемент теневого дерева" ("shadow root") к специальному элементу (и также к элементам `<div>`, `<span>`, `<body>`, `<article>`, `<main>`, `<nav>`, `<header>`, `<footer>`, `<section>`, `<p>`, `<blockquote>`, `<aside>` или `<h1>` — `<h6>`), который становится "ведущим элементом теневого дерева" ("shadow host"). Ведущие элементы теневого дерева, как и все HTML-элементы, уже являются корнем нормального дерева DOM из элементов потомков и текстовых узлов. Корневой элемент теневого дерева — это корень другого, более закрытого дерева из элементов потомков, которое произрастает из ведущего элемента теневого дерева и может считаться отдельным мини-документом.

Слово "теневая" в формулировке "теневая модель DOM" относится к тому факту, что элементы, которые происходят от корневого элемента теневого дерева, "прячутся в тени": они не являются частью нормального дерева DOM, не присутствуют в массиве `children` своего ведущего элемента и не посещаются методами обхода нормального дерева DOM, такими как `querySelector()`. Ради контраста дочерние элементы нормального дерева DOM ведущего элемента теневого дерева иногда называют "световой моделью DOM".

Чтобы понять назначение теневой модели DOM, представьте себе HTML-элементы `<audio>` и `<video>`: они отображают нетривиальный пользовательский интерфейс для управления воспроизведением мультимедиа, но кнопки воспроизведения и паузы и другие элементы пользовательского интерфейса не входят в состав дерева DOM и не могут изменяться в коде JavaScript. С учетом того, что веб-браузеры предназначены для отображения HTML-разметки, поставщики браузеров вполне естественно пожелают отображать внутренние пользовательские интерфейсы такого рода с применением HTML. На самом деле большинство браузеров давно делают нечто подобное, и теневая модель DOM превращает это в стандартную часть веб-платформы.

Инкапсуляция теневой модели DOM

Ключевой особенностью теневой модели DOM является обеспечиваемая ею инкапсуляция. Потомки корневого элемента теневого дерева скрыты — и независимы — от обыкновенного дерева DOM, почти как если бы они находились в независимом документе. Теневая модель DOM предоставляет три очень важных вида инкапсуляции.

- Как уже упоминалось, элементы в теневой модели DOM скрыты от методов обыкновенной модели DOM вроде `querySelectorAll()`. Когда корневой элемент теневого дерева создается и прикрепляется к своему

ведущему элементу теневого дерева, он может быть создан в "открытом" или "закрытом" режиме. Закрытый корневой элемент теневого дерева полностью запечатан и недоступен. Однако чаще корневые элементы теневых деревьев создаются в "открытом" режиме, а это значит, что ведущий элемент теневого дерева имеет свойство `shadowRoot`, которое код JavaScript может использовать для получения доступа к элементам корня теневого дерева, если на то есть причина.

- Стили, определенные ниже корневого элемента теневого дерева являются закрытыми по отношению к данному дереву, и они никогда не будут влиять на элементы световой модели DOM снаружи. (Корневой элемент теневого дерева может определять стандартные стили для своего ведущего элемента теневого дерева, но они будут переопределены стилями световой модели DOM.) Аналогично стили световой модели DOM, которые применяются к ведущему элементу теневого дерева, не воздействуют на потомков корневого элемента теневого дерева. Элементы в теневой модели DOM будут наследовать от световой модели DOM такие вещи, как размер шрифта и цвет фона, а стили в теневой модели DOM могут использовать переменные CSS, определенные в световой модели DOM. Но по большей части стили световой модели DOM и стили теневой модели DOM полностью независимы: автору веб-компонента и пользователю веб-компонента не придется беспокоиться о противоречиях или конфликтах между их таблицами стилей. Возможность "установления области видимости" CSS подобным образом — вероятно наиболее важная особенность теневой модели DOM.

- Определенные событие (вроде `"load"`), которые возникают внутри теневой модели DOM, ограничиваются теневой моделью DOM. Другие, включая события фокуса, мыши и клавиатуры, поднимаются подобно пузырькам вверх и наружу. Когда событие, произошедшее в теневой модели DOM, пересекает границу и начинает распространяться в световой модели DOM, его свойство `target` изменяется на ведущий элемент теневого дерева, поэтому кажется, что оно возникло прямо в этом элементе.

Слоты теневой модели DOM и дочерние элементы световой модели DOM

HTML-элемент, который является ведущим элементом теневого дерева, имеет два дерева потомков. Одно из них, хранящееся в массиве `children[]`, включает потомков обычной световой модели DOM ведущего элемента, а другое содержит корневой элемент теневого дерева и всех его потомков, и вас наверняка интересует, каким образом два отдельных дерева содержимого могут отображаться внутри того же самого ведущего элемента. Ниже описано, как все работает.

- Потомки корневого элемента теневого дерева всегда отображаются внутри ведущего элемента теневого дерева.

- Когда эти потомки включают элемент <slot>, тогда дочерние элементы обыкновенной световой модели DOM ведущего элемента отображаются, как если бы они были дочерними элементами данного <slot>, замещая любое содержимое теневой модели DOM в слоте. Если теневая модель DOM не включает <slot>, то любое содержимое световой модели DOM ведущего элемента никогда не отображается. Если теневая модель DOM имеет <slot>, но какие-либо дочерние элементы световой DOM в ведущем элементе теневого дерева отсутствуют, тогда содержимое теневой модели DOM слота отображается, как принято по умолчанию.
- Когда содержимое световой модели DOM отображается внутри слота теневой модели DOM, мы говорим, что эти элементы были "распространены", но важно понимать, что на самом деле элементы не стали частью теневой модели DOM. Их все еще можно запрашивать с помощью `querySelector()`, и они по-прежнему находятся в световой модели DOM как дочерние элементы или потомки ведущего элемента.
- Если в теневой модели DOM определено более одного <slot> и слоты именованы посредством атрибута name, тогда дочерние элементы ведущего элемента теневого дерева могут указывать, в каком слоте они хотели бы появиться, задавая атрибут `slot="имя_слота"`. Пример такого использования приводился в подразделе 15.6.1 во время демонстрации настройки значков, отображаемых компонентом <search-box>.

API-интерфейс теневой модели DOM

Несмотря на всю свою мощь, теневая модель DOM не располагает обширным API-интерфейсом для JavaScript. Чтобы превратить элемент световой модели DOM в ведущий элемент теневого дерева, просто вызовите его метод `attachShadow()`, передав `{mode:"open"}` в единственном аргументе. Метод `attachShadow()` возвращает объект корневого элемента теневого дерева и также устанавливает этот объект в качестве значения свойства `shadowRoot` ведущего элемента. Корневой элемент теневого дерева представляет собой объект `DocumentFragment`, и вы можете применять методы DOM для добавления к нему содержимого или просто установить его свойство `innerHTML` в строку HTML-разметки.

Если вашему веб-компоненту необходимо знать, когда изменилось содержимое световой модели DOM элемента <slot> теневой модели DOM, то он может зарегистрировать прослушиватель для событий `"slotchanged"` непосредственно в элементе <slot>.

15.6.5. Пример: веб-компонент <search-box>

Веб-компонент <search-box> был представлен на рис. 15.3. В примере 15.3 демонстрируются три технологии, которые делают возможным определение веб-компонентов: код реализует компонент <search-box> как специальный элемент, который использует дескриптор <template> для эффективности и корневой элемент теневого дерева для инкапсуляции.

В примере 15.3 показано, как работать с низкоуровневыми API-интерфейсами веб-компонентов напрямую. На практике многие разработанные в наши дни веб-компоненты создают их с применением библиотек более высокого уровня, таких как lit-element. Одна из причин использования библиотеки связана с тем, что успешно создавать многократно применяемые и настраиваемые компоненты на самом деле довольно трудно, и есть много деталей, которые должны делаться правильно. В примере 15.3 демонстрируется веб-компонент и реализована базовая обработка клавиатурного фокуса, но в остальном доступность игнорируется и не предпринимается никаких попыток использовать надлежащие атрибуты ARIA (Accessible Rich Internet Applications — доступные насыщенные Интернет-приложения), чтобы обеспечить работу компонента с читателями экрана и другими оказывающими помощь технологиями.

Пример 15.3. Реализация веб-компонента

```
/**
 * Этот класс определяет специальный HTML-элемент <search-box>, который отображает
 * поле ввода текста <input> плюс два значка или эмотикона. По умолчанию он
 * отображает эмотикон увеличительного стекла (указывающий на поиск) слева
 * от текстового поля и эмотикон X (указывающий на отмену) справа от текстового
 * поля. Он скрывает границу поля ввода и отображает границу вокруг себя, создавая
 * видимость, что два эмотикона находятся внутри поля ввода. Аналогично, когда
 * внутреннее поле ввода находится в фокусе, то вокруг <search-box>
 * отображается фокальное кольцо.
 *
 * Вы можете переопределять стандартные значки, включая дочерние
 * элементы <span> или <img> элемента <search-box> с помощью
 * атрибутов slot="left" и slot="right".
 *
 * <search-box> поддерживает нормальные HTML-атрибуты disabled и hidden,
 * а также атрибуты size и placeholder, которые имеют в этом элементе
 * тот же смысл, что и в элементе <input>.
 *
 * События ввода из внутреннего элемента <input> поднимаются пузырьком вверх и
 * появляются со своими свойствами target, установленными в элемент <search-box>.
 *
 * Элемент инициирует событие "search" со свойством detail, установленным
 * в текущую введенную строку, когда пользователь щелкает на левом эмотиконе
 * (увеличительном стекле). Событие "search" также посылается, когда
 * внутреннее текстовое поле генерирует событие "change" (когда текст
 * изменился и пользователь нажал клавишу <Return> или <Tab>).
 *
 * Элемент инициирует событие "clear", когда пользователь щелкает на правом
 * эмотиконе (X). Если ни один из обработчиков не вызывает preventDefault()
 * для этого события, тогда элемент очищает ввод пользователя после
 * завершения отправки события.
 *
 * Обратите внимание, что свойства или атрибуты типа onsearch и onclear
 * отсутствуют: обработчики для событий "search" и "clear" могут
 * регистрироваться только посредством addEventListener().
 */
```

```
class SearchBox extends HTMLElement {
  constructor() {
    super();  // Вызов конструктора суперкласса; обязан быть первым.

  // Создать теневое дерево DOM и прикрепить его к этому элементу,
  // устанавливая значение свойства this.shadowRoot.
  this.attachShadow({mode: "open"});

  // Клонировать шаблон, который определяет потомков и таблицу стилей
  // для этого специального элемента, и добавить такое содержимое
  // к корневому элементу теневого дерева.
  this.shadowRoot.append(SearchBox.template.content.cloneNode(true));

  // Получить ссылки на важные элементы в теневой модели DOM.
  this.input = this.shadowRoot.querySelector("#input");
  let leftSlot = this.shadowRoot.querySelector('slot[name="left"]');
  let rightSlot = this.shadowRoot.querySelector('slot[name="right"]');

  // Когда внутреннее поле ввода получает или теряет фокус, установить
  // или удалить атрибут "focused", что заставит нашу внутреннюю таблицу
  // стилей отобразить или скрыть поддельное фокальное кольцо для целого
  // компонента. Обратите внимание, что события "blur" и "focus" совершают
  // пузырьковый подъем и выглядят происходящими в элементе <search-box>.
  this.input.onfocus = () => { this.setAttribute("focused", ""); };
  this.input.onblur = () => { this.removeAttribute("focused");};

  // Если пользователь щелкает на увеличительном стекле, тогда генерируется
  // событие "search". Оно также инициируется, если поле ввода генерирует
  // событие "change". (Событие "change" не поднимается пузырьком
  // за пределы теневой модели DOM.)
  leftSlot.onclick = this.input.onchange = (event) => {
    event.stopPropagation(); // Предотвратить подъем пузырьком событий щелчка
    if (this.disabled) return; // Ничего не делать, когда элемент отключен.
      this.dispatchEvent(new CustomEvent("search", {
        detail: this.input.value
      }));
    };

    // Если пользователь щелкает на X, тогда генерируется событие "clear".
    // Если метод preventDefault() не вызывался для события, очистить ввод.
    rightSlot.onclick = (event) => {
      event.stopPropagation(); // Не позволять событию щелчка подниматься
      if (this.disabled) return;// Ничего не делать, когда элемент отключен
      let e = new CustomEvent("clear", { cancelable: true });
      this.dispatchEvent(e);
      if (!e.defaultPrevented) { // Если событие не было "отменено",
        this.input.value = "";  // тогда очистить поле ввода.
      }
    };
  }

  // Когда некоторые из наших атрибутов устанавливаются или изменяются,
  // мы должны установить соответствующее значение во внутреннем элементе
  // <input>. Об этом позаботится данный метод жизненного цикла вместе
  // с определенным ниже статическим свойством observedAttributes.
  attributeChangedCallback(name, oldValue, newValue) {
```

```
        if (name === "disabled") {
            this.input.disabled = newValue !== null;
        } else if (name === "placeholder") {
            this.input.placeholder = newValue;
        } else if (name === "size") {
            this.input.size = newValue;
        } else if (name === "value") {
            this.input.value = newValue;
        }
    }

    // В заключение мы определяем методы получения и установки для свойств,
    // которые соответствуют поддерживаемым нами HTML-атрибутам. Методы получения
    // просто возвращают значение (или наличие) атрибута, а методы установки
    // всего лишь устанавливают значение (или наличие) атрибута. Когда метод
    // установки изменяет атрибут, браузер автоматически вызовет определенный
    // выше attributeChangedCallback().
    get placeholder() { return this.getAttribute("placeholder"); }
    get size() { return this.getAttribute("size"); }
    get value() { return this.getAttribute("value"); }
    get disabled() { return this.hasAttribute("disabled"); }
    get hidden() { return this.hasAttribute("hidden"); }

    set placeholder(value) { this.setAttribute("placeholder", value); }
    set size(value) { this.setAttribute("size", value); }
    set value(text) { this.setAttribute("value", text); }
    set disabled(value) {
        if (value) this.setAttribute("disabled", "");
        else this.removeAttribute("disabled");
    }
    set hidden(value) {
        if (value) this.setAttribute("hidden", "");
        else this.removeAttribute("hidden");
    }
}

// Это статическое поле требуется для метода attributeChangedCallback().
// Инициировать вызовы данного метода будут только атрибуты,
// имена которых указаны в этом массиве.
SearchBox.observedAttributes = ["disabled", "placeholder", "size", "value"];

// Создать элемент <template> для хранения таблицы стилей и дерева элементов,
// которые мы будем использовать для каждого экземпляра элемента SearchBox.
SearchBox.template = document.createElement("template");

// Мы инициализируем шаблон, синтаксически анализируя эту строку HTML-разметки.
// Однако обратите внимание, что когда мы создаем экземпляр SearchBox,
// то можем просто клонировать узлы в шаблоне и нам придется снова проводить
// синтаксический анализ HTML-разметки.
SearchBox.template.innerHTML = `
<style>
```

```
/*
 * Селектор :host ссылается на элемент <search-box> в световой модели DOM.
 * Эти стили применяются по умолчанию и могут быть переопределены
 * пользователем элемента <search-box> стилями в световой модели DOM.
 */
:host {
  display: inline-block;    /* Встроенное отображение по умолчанию. */
  border: solid black 1px;  /* Рамка со скругленными углами вокруг <input>
                               и <slot>. */
  border-radius: 5px;
  padding: 4px 6px;         /* Некоторое пространство внутри границы. */
}
:host([hidden]) {           /* Обратите внимание на круглые скобки:
                               когда ведущий элемент скрыт... */
  display:none;             /* ...набор атрибутов не отображает его. */
}
:host([disabled]) {         /*Когда ведущий элемент имеет атрибут disabled...*/
  opacity: 0.5;             /* ...сделать его полупрозрачным. */
}
:host([focused]) {          /*Когда ведущий элемент имеет атрибут focused...*/
  box-shadow: 0 0 2px 2px #6AE; /*...отобразить это фиктивное фокальное кольцо*/
}

/*Остаток таблицы стилей применяется только к элементам в теневой модели DOM.*/
input {
  border-width: 0;          /* Скрыть границу внутреннего поля ввода. */
  outline: none;            /* Также скрыть фокальное кольцо. */
  font: inherit;            /*По умолчанию элементы <input> не наследуют шрифт*/
  background: inherit;      /* То же относится к цвету фона. */
}
slot {
  cursor: default;          /* Указатель в форме стрелки над кнопками. */
  user-select: none;        /* Не разрешать пользователю выбирать текст
                               эмотикона. */
}
</style>

<div>
  <slot name="left">\u{1f50d}</slot>     <!-- U+1F50D - эмотикон с
                                              увеличительным стеклом -->
  <input type="text" id="input" />       <!-- Фактический элемент ввода -->
  <slot name="right">\u{2573}</slot>     <!-- U+2573 - эмотикон с X -->
</div>
`;

// Наконец, мы вызываем customElement.define() для регистрации
// элемента SearchBox в качестве реализации дескриптора <search-box>.
// Специальные элементы должны иметь имя дескриптора, которое содержит дефисы.
customElements.define("search-box", SearchBox);
```

15.7. SVG: масштабируемая векторная графика

Масштабируемая векторная графика (scalable vector graphics — SVG) является форматом изображений. Слово "векторная" в названии служит признаком на то, что он фундаментально отличается от форматов растровых изображений, таких как GIF, JPEG и PNG, которые указывают матрицу значений пикселей. Напротив, "изображение" SVG представляет собой точное, независимое от разрешения (отсюда "масштабируемая") описание шагов, необходимых для вычерчивания желаемой графики. Изображения SVG описываются посредством текстовых файлов, с применением языка разметки XML, который довольно похож на HTML.

Существуют три способа использования SVG в веб-браузерах.

1. Вы можете применять файлы изображений `.svg` с обыкновенными HTML-дескрипторами `<img>`, как если бы использовали изображение `.png` или `.jpeg`.
2. Поскольку основанный на XML формат SVG настолько похож на HTML, в действительности вы можете встраивать дескрипторы SVG прямо в свои HTML-документы. В таком случае синтаксический анализатор HTML браузера позволяет опускать пространства имен XML и обращаться с дескрипторами SVG, как если бы они были HTML-дескрипторами.
3. Вы можете применять API-интерфейс DOM для динамического создания элементов SVG, чтобы генерировать изображения по запросу.

В последующих разделах демонстрируются второй и третий способы использования SVG. Тем не менее, обратите внимание, что SVG имеет широкую и умеренно сложную грамматику. Помимо простых примитивов для вычерчивания фигур она включает поддержку произвольных кривых, текста и анимации. Графика SVG может даже содержать в себе сценарии JavaScript и таблицы стилей CSS для добавления линий поведения и информации о представлении. Полное описание SVG выходит далеко за рамки настоящей книги. Цель этого раздела — просто показать вам, как можно применять графику SVG в своих HTML-документах и работать с ней в коде JavaScript.

15.7.1. SVG в HTML

Разумеется, изображения SVG можно отображать с использованием HTML-дескрипторов `<img>`. Но вы также можете встраивать графику SVG напрямую в HTML-разметку. Делая это, вы даже можете применять таблицы стилей CSS для указания таких вещей, как шрифты, цвета и ширины линий. Ниже приведено содержимое HTML-файла, в котором используется SVG для отображения циферблата аналоговых часов:

```
<html>
<head>
<title>Analog Clock</title>
<style>
```

```
/* Все эти стили CSS применяются к элементам SVG, определенным далее. */
#clock {                          /* Стили для всего в часах:*/
   stroke: black;                 /* черные линии */
   stroke-linecap: round;         /* со скругленными концами */
   fill: #ffe;                    /* на желтоватом фоне. */
}
#clock .face { stroke-width: 3; }    /* Эскиз циферблата часов. */
#clock .ticks { stroke-width: 2; } /* Линии, которые помечают каждый час*/
#clock .hands { stroke-width: 3; }   /* Как рисовать стрелки часов. */
#clock .numbers {                    /* Как рисовать числа. */
   font-family: sans-serif; font-size: 10; font-weight: bold;
   text-anchor: middle; stroke: none; fill: black;
}
</style>
</head>
<body>
  <svg id="clock" viewBox="0 0 100 100" width="250" height="250">
    <!-- Атрибуты width и height - это экранный размер графики. -->
    <!-- Атрибут viewBox поддерживает внутреннюю систему координат. -->
    <circle class="face" cx="50" cy="50" r="45"/> <!-- Циферблат. -->
    <g class="ticks">    <!-- Отметки для каждого из 12 часов. -->
      <line x1='50' y1='5.000' x2='50.00' y2='10.00'/>
      <line x1='72.50' y1='11.03' x2='70.00' y2='15.36'/>
      <line x1='88.97' y1='27.50' x2='84.64' y2='30.00'/>
      <line x1='95.00' y1='50.00' x2='90.00' y2='50.00'/>
      <line x1='88.97' y1='72.50' x2='84.64' y2='70.00'/>
      <line x1='72.50' y1='88.97' x2='70.00' y2='84.64'/>
      <line x1='50.00' y1='95.00' x2='50.00' y2='90.00'/>
      <line x1='27.50' y1='88.97' x2='30.00' y2='84.64'/>
      <line x1='11.03' y1='72.50' x2='15.36' y2='70.00'/>
      <line x1='5.000' y1='50.00' x2='10.00' y2='50.00'/>
      <line x1='11.03' y1='27.50' x2='15.36' y2='30.00'/>
      <line x1='27.50' y1='11.03' x2='30.00' y2='15.36'/>
    </g>
    <g class="numbers">  <!-- Нумерация основных направлений. -->
      <text x="50" y="18">12</text><text x="85" y="53">3</text>
      <text x="50" y="88">6</text><text x="15" y="53">9</text>
    </g>
    <g class="hands"> <!-- Вычертить стрелки, указывающие прямо вверх-->
      <line class="hourhand" x1="50" y1="50" x2="50" y2="25"/>
      <line class="minutehand" x1="50" y1="50" x2="50" y2="20"/>
    </g>
  </svg>
  <script src="clock.js"></script>
</body>
</html>
```

Вы наверняка заметили, что потомки дескриптора <svg> не являются нормальными HTML-дескрипторами. Однако дескрипторы <circle>, <line> и <text> имеют очевидные целевые назначения, и должно быть ясно, как работает такая графика SVG. Тем не менее, существует много других дескрипторов SVG, и за дополнительными сведениями о них вам придется обращаться в спра-

вочник по SVG. Вы могли также отметить странность таблицы стилей. Стили вроде `fill`, `stroke-width` и `text-anchor` не относятся к нормальным свойствам стилей CSS. В данном случае CSS по существу применяется для установки атрибутов дескрипторов SVG, которые присутствуют в документе. Кроме того, обратите внимание, что сокращенное свойство CSS по имени `font` для дескрипторов SVG не работает, а потому вы должны явно устанавливать `font-family`, `font-size` и `font-weight` как отдельные свойства стиля.

15.7.2. Работа с SVG в сценариях

Одна из причин встраивания графики SVG прямо в HTML-файлы (вместо использования статических дескрипторов `<img>`) связана с тем, что в таком случае вы можете применять API-интерфейс DOM для манипулирования изображением SVG. Предположим, что вы используете SVG, чтобы отображать значки в своем веб-приложении. Вы могли бы встроить SVG в дескриптор `<template>` (см. подраздел 15.6.2) и затем клонировать содержимое шаблона всякий раз, когда необходимо вставить копию значка в пользовательский интерфейс. А если вы хотите, чтобы значок реагировал на пользовательскую активность, скажем, изменяя цвет при наведении на него указателя мыши, то добиться этого часто удается с помощью CSS.

Графикой SVG, непосредственно встроенной в HTML-разметку, можно также динамически манипулировать. В рассмотренном ранее примере с аналоговыми часами отображался статический циферблат с часовой и минутной стрелками, которые были направлены прямо вверх, показывая полночь или полдень. Но вы могли заметить, что HTML-файл включает дескриптор `<script>`. Данный сценарий периодически запускает функцию для проверки времени и трансформации часовой и минутной стрелок за счет их поворота на соответствующее количество градусов, чтобы часы фактически отображали текущее время, как показано на рис. 15.5. Код для манипулирования изображением часов прямолинеен. Он определяет надлежащий угол часовой и минутной стрелок на основе текущего времени, затем применяет `querySelector()`, чтобы найти элементы SVG, которые отображают эти стрелки, и в заключение устанавливает для них атрибут `transform`, чтобы выполнить поворот относительно центра циферблата.

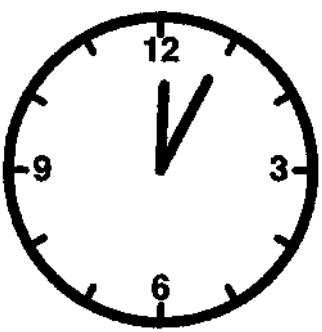


Рис. 15.5. Аналоговые часы, созданные за счет манипулирования SVG в сценарии

Функция использует setTimeout() для обеспечения своего запуска один раз в минуту:

```
(function updateClock() {   // Обновляет изображение SVG часов,
                            // чтобы показывать текущее время.
    let now = new Date();                       // Текущее время
    let sec = now.getSeconds();                 // Секунды
    let min = now.getMinutes() + sec/60;        // Дробные минуты
    let hour = (now.getHours() % 12) + min/60; // Дробные часы
    let minangle = min * 6;                     // 6 градусов на минуту
    let hourangle = hour * 30;                  // 30 градусов на час

    // Получить элементы SVG для стрелок часов.
    let minhand = document.querySelector("#clock .minutehand");
    let hourhand = document.querySelector("#clock .hourhand");

    // Установить для них атрибут SVG, чтобы перемещать по циферблату.
    minhand.setAttribute("transform", `rotate(${minangle},50,50)`);
    hourhand.setAttribute("transform", `rotate(${hourangle},50,50)`);

    // Запустить эту функцию снова через 10 секунд.
    setTimeout(updateClock, 10000);
}()); // Обратите внимание на немедленный вызов этой функции.
```

15.7.3. Создание изображений SVG с помощью JavaScript

В дополнение к простому манипулированию в коде изображениями SVG, встроенными в ваши HTML-документы, вы также можете создавать изображения SVG с нуля, что полезно, например, при визуализации динамически загружаемых данных. В примере 15.4 демонстрируется применение JavaScript для создания круговых диаграмм SVG, подобных показанной на рис. 15.6.

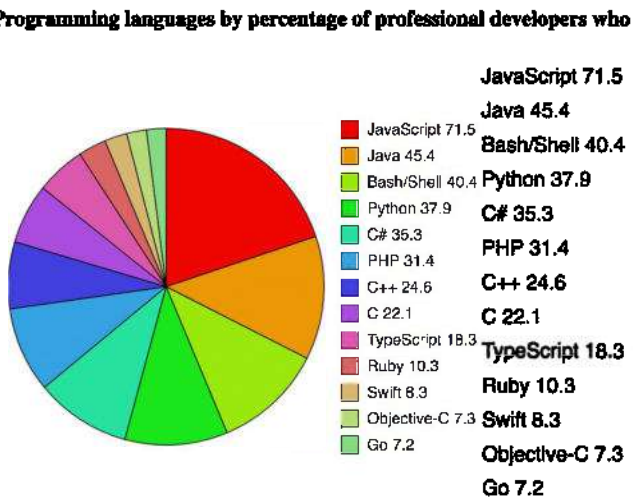


Рис. 15.6. Круговая диаграмма SVG, построенная посредством JavaScript (данные опроса разработчиков на предмет самых популярных технологий Stack Overflow 2018)

Несмотря на то что дескрипторы SVG можно включать в HTML-документы, формально они являются дескрипторами XML, а не HTML, и если вы хотите создавать элементы SVG с помощью API-интерфейса DOM для JavaScript, то не можете использовать обычную функцию `createElement()`, представленную в подразделе 15.3.5. Взамен вам придется применять функцию `createElementNS()`, которая принимает в своем первом аргументе строку с пространством имен XML. Для SVG пространством имен будет литеральная строка `"http://www.w3.org/2000/svg"`.

За исключением использования `createElementNS()` код вычерчивания круговой диаграммы в примере 15.4 относительно прямолинеен. В нем присутствует немного математических расчетов для преобразования отображаемых данных в углы секторов круговой диаграммы. Однако основной объем примера занимает код DOM, который создает элементы SVG и устанавливает их атрибуты.

Самая трудная для понимания часть примера — код, который вычерчивает фактические секторы круговой диаграммы. Для отображения каждого сектора применяется элемент `<path>`. Этот элемент SVG описывает произвольные фигуры, состоящие из прямых и кривых линий. Описание фигуры задается атрибутом `d` элемента `<path>`. Внутри значения атрибута `d` используется компактная грамматика из буквенных кодов и чисел, которые определяют координаты, углы и другие величины. Скажем, буква M означает "move to" (перейти к) и за ней следуют координаты *x* и *y*. Буква L означает "line to" (провести линию к) и обеспечивает вычерчивание линии от текущей точки до указанных за буквой координат. В приведенном примере также применяется буква A для вычерчивания дуги (arc). За ней следуют семь чисел, описывающих дугу, и при желании вы можете ознакомиться с синтаксисом на онлайновых ресурсах.

Пример 15.4. Вычерчивание круговой диаграммы с помощью JavaScript и SVG

```
/**
 * Создает элемент <svg> и вычерчивает внутри него круговую диаграмму.
 *
 * Эта функция ожидает объектный аргумент со следующими свойствами:
 *
 *   width, height: размер графики SVG в пикселях
 *   cx, cy, r: центр и радиус сектора
 *   lx, ly: левый верхний угол условных обозначений диаграммы
 *   data: объект, имена свойств которого являются метками данных,
 *         а значения свойств - значениями, ассоциированными с метками
 *
 * Функция возвращает элемент <svg>. Вызывающий код должен вставить
 * его в документ, чтобы сделать видимым.
 */
function pieChart(options) {
    let {width, height, cx, cy, r, lx, ly, data} = options;

    // Пространство имен XML для элементов svg.
    let svg = "http://www.w3.org/2000/svg";

    // Создать элемент <svg> и указать размер пикселя
    // и пользовательские координаты.
```

```
let chart = document.createElementNS(svg, "svg");
chart.setAttribute("width", width);
chart.setAttribute("height", height);
chart.setAttribute("viewBox", `0 0 ${width} ${height}`);

// Определить стили текста, которые будут использоваться в диаграмме.
// Если мы не установим эти значения здесь,
// тогда их можно установить с помощью CSS.
chart.setAttribute("font-family", "sans-serif");
chart.setAttribute("font-size", "18");

// Получить метки и значения в виде массивов и сложить значения,
// чтобы узнать, насколько велика круговая диаграмма.
let labels = Object.keys(data);
let values = Object.values(data);
let total = values.reduce((x,y) => x+y);

// Вычислить углы для всех секторов. Сектор i начинается в angles[i]
// и заканчивается в angles[i+1]. Углы измеряются в радианах.
let angles = [0];
values.forEach((x, i) => angles.push(angles[i] + x/total * 2 * Math.PI));

// Организовать цикл по секторам круговой диаграммы.
values.forEach((value, i) => {
    // Рассчитать две точки, где наш сектор пересекается с окружностью.
    // Эти формулы выбраны так, чтобы угол 0 соответствовал 12 часам
    // и положительные углы увеличивались по часовой стрелке.
    let x1 = cx + r * Math.sin(angles[i]);
    let y1 = cy - r * Math.cos(angles[i]);
    let x2 = cx + r * Math.sin(angles[i+1]);
    let y2 = cy - r * Math.cos(angles[i+1]);

    // Это флаг для углов, превышающих половину окружности.
    // Он требуется для компоненты вычерчивания arc в SVG.
    let big = (angles[i+1] - angles[i] > Math.PI) ? 1 : 0;

    // Эта строка описывает, как вычерчивать сектор круговой диаграммы:
    let path = `M${cx},${cy}` +    // Перейти в центр окружности.
        `L${x1},${y1}` +           // Вычертить линию до (x1,y1).
        `A${r},${r} 0 ${big} 1` +  // Вычертить дугу радиуса r...
        `${x2},${y2}` +            // ...закончив ее в (x2,y2).
        "Z";                       // Замкнуть путь в (cx,cy).

    // Вычислить цвет CSS для этого сектора. Формула работает только
    // для примерно 15 цветов.
    // Таким образом, не включайте в диаграмму более 15 секторов.
    let color = `hsl(${(i*40)%360},${90-3*i}%,${50+2*i}%)`;

    // Мы описываем сектор с помощью элемента <path>.
    // Обратите внимание на createElementNS().
    let slice = document.createElementNS(svg, "path");

    // Установить атрибуты элемента <path>.
    slice.setAttribute("d", path);   // Установить путь для этого сектора.
    slice.setAttribute("fill", color);          // Установить цвет сектора.
    slice.setAttribute("stroke", "black");      // Сделать контур черным.
    slice.setAttribute("stroke-width", "1");    // Ширина 1 пиксель CSS.
    chart.append(slice);              // Добавить сектор в диаграмму.
```

```
        // Вычертить небольшой квадрат, соответствующий ключу.
        let icon = document.createElementNS(svg, "rect");
        icon.setAttribute("x", lx);              // Позиция квадрата.
        icon.setAttribute("y", ly + 30*i);
        icon.setAttribute("width", 20);          // Размер квадрата.
        icon.setAttribute("height", 20);
        icon.setAttribute("fill", color);        // Такой же цвет заполнения,
                                                 // как у сектора.
        icon.setAttribute("stroke", "black"); // Такой же контур.
        icon.setAttribute("stroke-width", "1");
        chart.append(icon);                      // Добавить в диаграмму.

        // Добавить метку справа от квадрата.
        let label = document.createElementNS(svg, "text");
        label.setAttribute("x", lx + 30);        // Позиция текста.
        label.setAttribute("y", ly + 30*i + 16);
        label.append(`${labels[i]} ${value}`);// Добавить текст к метке.
        chart.append(label);                     // Добавить метку в диаграмму.
    });
    return chart;
}
```

Круговая диаграмма на рис. 15.6 была создана с использованием функции `pieChart()` из примера 15.4:

```
document.querySelector("#chart").append(pieChart({
    width: 640, height:400,      // Общий размер диаграммы.
    cx: 200, cy: 200, r: 180,    // Центр и радиус круговой диаграммы.
    lx: 400, ly: 10,             // Позиция условных обозначений.
    data: {                      // Данные для построения диаграммы.
        "JavaScript": 71.5,
        "Java": 45.4,
        "Bash/Shell": 40.4,
        "Python": 37.9,
        "C#": 35.3,
        "PHP": 31.4,
        "C++": 24.6,
        "C": 22.1,
        "TypeScript": 18.3,
        "Ruby": 10.3,
        "Swift": 8.3,
        "Objective-C": 7.3,
        "Go": 7.2,
    }
}));
```

15.8. Графика в <canvas>

Элемент <canvas> не обладает собственным внешним видом, но создает внутри документа поверхность рисования и предоставляет мощный API-интерфейс рисования для JavaScript стороны клиента. Главное отличие API-интерфейса

<canvas> и SVG связано с тем, что в случае холста рисунки создаются за счет вызова методов, а в случае SVG — путем построения дерева элементов XML. Эти два подхода одинаково эффективны: один может эмулироваться посредством другого. Тем не менее, на первый взгляд они выглядят совершенно разными, и у каждого есть свои сильные и слабые стороны. Скажем, рисунок SVG легко редактировать, удаляя элементы из его описания. Чтобы удалить элемент из той же самой графики в <canvas>, часто необходимо очистить рисунок и нарисовать его с нуля. Поскольку API-интерфейс рисования Canvas основан на JavaScript и относительно компактен (в отличие от грамматики SVG), он документируется в книге более подробно.

Трехмерная графика в Canvas

Вы также можете вызвать `getContext()` со строкой `"webgl"`, чтобы получить объект контекста, который даст возможность рисовать трехмерную графику с применением API-интерфейса WebGL. Крупный, сложный и низкоуровневый API-интерфейс WebGL позволяет программистам на JavaScript обращаться к графическому процессору, писать специальные шейдеры (т.е. затеняющие программы) и выполнять другие очень мощные графические операции. Однако WebGL в этой книге не рассматривается: разработчики веб-приложений больше предпочитают использовать служебные библиотеки, построенные поверх WebGL, нежели работать с API-интерфейсом WebGL напрямую.

Большая часть API-интерфейса рисования Canvas определена не в самом элементе <canvas>, а в объекте "контекста рисования", получаемом с помощью метода `getContext()` холста. Вызовите `getContext()` с аргументом `"2d"`, чтобы получить объект `CanvasRenderingContext2D`, который можно применять для рисования двумерной графики внутри холста.

В качестве простого примера использования API-интерфейса Canvas взгляните на следующий HTML-документ, где посредством элементов <canvas> и кода JavaScript отображаются две простые фигуры:

```
<p>This is a red square: <canvas id="square" width=10 height=10>
</canvas>.
<p>This is a blue circle: <canvas id="circle" width=10 height=10>
</canvas>.
<script>
let canvas=document.querySelector("#square"); // Получить первый
                                              // элемент canvas.
let context = canvas.getContext("2d");        // Получить контекст для
                                              // двумерного рисования.
context.fillStyle = "#f00";                   // Установить цвет
                                              // заполнения в красный.
context.fillRect(0,0,10,10);                  // Заполнить квадрат.
canvas = document.querySelector("#circle"); // Получить второй
                                            // элемент canvas.
```

```
context = canvas.getContext("2d");              // Получить его контекст.
context.beginPath();                            // Начать новый "путь".
context.arc(5, 5, 5, 0, 2*Math.PI, true);       // Добавить окружность
                                                // к пути.
context.fillStyle = "#00f";                     // Установить цвет
                                                // заполнения в синий.
context.fill();                                 // Заполнить путь.
</script>
```

Вы уже видели, что сложные фигуры в SVG описываются как "пути" прямых и кривых линий, которые можно чертить или заполнять. В API-интерфейсе Canvas также применяется понятие пути. Вместо описания пути как строки букв и чисел путь определяется последовательностью вызовов методов наподобие вызовов `beginPath()` и `arc()` в предыдущем коде. После того, как путь определен, с ним будут работать другие методы вроде `fill()`. Способ выполнения таких операций указывается через разнообразные свойства объекта контекста, например, `fillStyle`.

В последующих подразделах демонстрируются методы и свойства API-интерфейса Canvas для двумерной графики. В большинстве приводимых далее примеров кода задействована переменная `c`, в которой хранится объект `CanvasRenderingContext2D` холста, но код ее инициализации временами не показан. Для выполнения этих примеров вам придется добавить HTML-разметку, чтобы определить холст с надлежащими атрибутами `width` и `height`, и затем добавить код следующего вида, чтобы инициализировать переменную `c`:

```
let canvas = document.querySelector("#my_canvas_id");
let c = canvas.getContext('2d');
```

15.8.1. Пути и многоугольники

Чтобы начертить линии на холсте и заполнить области, охватываемые линиями, сначала необходимо определить *путь*, который представляет собой последовательность из одного или большего количества подпутей. Подпуть — это последовательность из двух и более точек, соединяемых отрезками прямой (или, как будет показано позже, отрезками кривой). Новый путь начинается с помощью метода `beginPath()`, а новый подпуть — посредством метода `moveTo()`. После установления стартовой точки подпути методом `moveTo()` можно соединить данную точку с новой точкой прямой линией, вызвав метод `lineTo()`. В приведенном ниже коде определяется путь, который включает два отрезка прямой:

```
c.beginPath();          // Начать новый путь.
c.moveTo(100, 100);     // Начать подпуть в точке (100,100).
c.lineTo(200, 200);     // Добавить линию из (100,100) в (200,200).
c.lineTo(100, 200);     // Добавить линию из (200,200) в (100,200).
```

В коде просто определяется путь; на холсте ничего не рисуется. Чтобы начертить два отрезка прямой в пути, нужно вызывать метод `stroke()`, а чтобы заполнить область, определенную такими отрезками прямой — метод `fill()`:

```
c.fill();               // Заполнить треугольную область.
c.stroke();             // Вычертить два ребра треугольника.
```

Такой код (вместе с дополнительным кодом для установки ширин линий и цветов заполнения) производит результат, показанный на рис. 15.7.

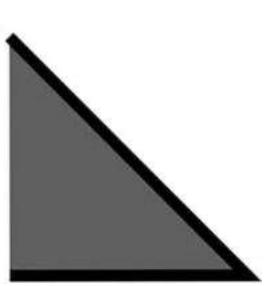


Рис. 15.7. Простой вычерченный и заполненный путь

Обратите внимание, что подпуть, представленный на рис. 15.7, "разомкнут". Он состоит всего лишь из двух отрезков прямой, и конечная точка не соединяется с начальной точкой. Это означает, что подпуть не охватывает область. Метод `fill()` заполняет разомкнутые подпути, действуя так, как если бы последняя точка в подпути соединялась с первой точкой. Вот почему код заполняет треугольник, но вычерчивает только два его ребра.

Если вы хотите вычертить все три ребра только что показанного треугольника, тогда должны вызвать метод `closePath()`, чтобы соединить конечную точку подпути с начальной точкой. (Вы могли бы также вызвать `lineTo(100,100)`, но тогда результатом будут три отрезка прямой, которые разделяют начальную и конечную точку, но не являются по-настоящему замкнутыми. При вычерчивании толстыми линиями визуальный результат будет лучше, если используется `closePath()`.)

Есть еще два важных момента, которые следует отметить относительно методов `stroke()` и `fill()`. Первый касается того, что оба метода оперируют на всех подпутях в текущем пути. Пусть мы добавили в предыдущий код дополнительный подпуть:

```
c.moveTo(300,100);//Начать новый подпуть в точке (300,100).
c.lineTo(300,200); //Начертить вертикальную линию вниз до точки (300,200)
```

Если бы мы затем вызвали `stroke()`, то вычертили бы два соединенных ребра треугольника и несоединенную вертикальную линию.

Второй момент, связанный с методами `stroke()` и `fill()`, заключается в том, что ни один из них не изменяет текущий путь: вы можете вызывать `fill()` и путь останется на месте, когда вы вызовите `stroke()`. Если вы завершили работу с одним путем и хотите начать другой, то должны не забывать о вызове `beginPath()`, иначе новые подпути будут добавляться к существующему пути, и в итоге может оказаться, что вы вычерчиваете старые подпути снова и снова.

В примере 15.5 определяется функция для рисования обыкновенных многоугольников и демонстрируется применение методов `moveTo()`, `lineTo()` и `closePath()` для определения подпутей, а также `fill()` и `stroke()` для вычерчивания этих путей. Результат его выполнения показан на рис. 15.8.

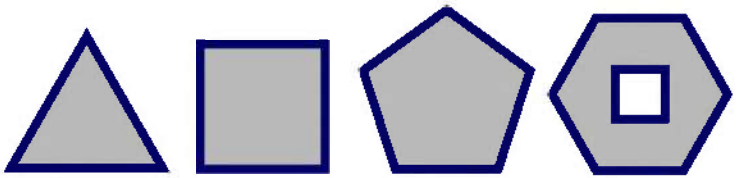


Рис. 15.8. Обыкновенные многоугольники

Пример 15.5. Обыкновенные многоугольники, вычерченные с помощью moveTo(), lineTo() и closePath()

```
// Определяет обыкновенный многоугольник с n сторон, радиусом r и центром
// в (x,y). Вершины равномерно распределены по окружности. Помещает первую
// вершину прямо вверху или под указанным углом. Поворачивает
// по часовой стрелке, если только последний аргумент не равен true.
function polygon(c, n, x, y, r, angle=0, counterclockwise=false) {
   c.moveTo(x + r*Math.sin(angle), // Начать новый подпуть в первой вершине.
            y - r*Math.cos(angle));  // Использовать тригонометрию
                                     // для расчета позиции.

   let delta = 2*Math.PI/n;          // Угловое расстояние между вершинами.
   for(let i = 1; i < n; i++) {      // Для каждой из оставшихся вершин.
      angle += counterclockwise?-delta:delta;     // Подогнать угол.
      c.lineTo(x + r*Math.sin(angle),             // Добавить прямую до
                                                  // следующей вершины.
               y - r*Math.cos(angle));
   }
   c.closePath();                   // Соединить последнюю вершину с первой.
}

// Предположить, что есть только один холст, и получить
// его объект контекста для вычерчивания.
let c = document.querySelector("canvas").getContext("2d");

// Начать новый путь и добавить подпути многоугольника.
c.beginPath();
polygon(c, 3, 50, 70, 50);                          // Треугольник
polygon(c, 4, 150, 60, 50, Math.PI/4);              // Квадрат
polygon(c, 5, 255, 55, 50);                         // Пятиугольник
polygon(c, 6, 365, 53, 50, Math.PI/6);              // Шестиугольник
polygon(c, 4, 365, 53, 20, Math.PI/4, true);        // Маленький квадрат
                                                    // внутри шестиугольника

// Установить ряд свойств, которые управляют тем, как будет выглядеть графика
c.fillStyle = "#ccc";               // Светло-серые внутренности,
c.strokeStyle = "#008";             // обведенные темно-синими линями
c.lineWidth = 5;                    // шириной пять пикселей.

// Вычертить все многоугольники (каждый в собственном подпути)
// с помощью следующих вызовов.
c.fill();                           // Заполнить фигуры
c.stroke();                         // и вычертить их контуры.
```

Обратите внимание, что в этом примере рисуется шестиугольник с квадратом внутри него. Квадрат и шестиугольник являются отдельными подпутями, но они перекрываются. Когда подобное происходит (или когда одиночный подпуть пересекает сам себя), холст должен иметь возможность определить, какая область находится внутри и какая снаружи. Для этого холст использует проверку, известную как "правило ненулевого индекса". В данном случае внутренняя часть квадрата не заполнена, потому что квадрат и шестиугольник вычерчивались в противоположных направлениях: вершины шестиугольника соединялись отрезками прямой, двигаясь по часовой стрелке, а вершины квадрата соединялись с движением против часовой стрелки. Если бы квадрат был вычерчен с движением тоже по часовой стрелке, тогда вызов `fill()` заполнил бы также и внутреннюю часть квадрата.

15.8.2. Размеры и координаты холста

Атрибуты `width` и `height` элемента `<canvas>` и соответствующие им свойства `width` и `height` объекта холста задают размеры холста. Стандартная система координат холста размещает начало (0,0) в левом верхнем углу холста. Координата *x* увеличивается вправо, а координата *y* — по мере движения к низу экрана. Координаты на холсте могут указываться с применением значений с плавающей точкой.

Размеры холста нельзя изменять без его полной переустановки. Установка любого из свойств `width` или `height` объекта холста (даже в текущее значение) приводит к очистке холста, стиранию текущего пути и сбросу всех атрибутов графики (включая текущую трансформацию и область отсечения) в первоначальное состояние.

Атрибуты `width` и `height` холста определяют фактическое количество пикселей, которые можно нарисовать на холсте. Для каждого пикселя выделяются четыре байта памяти, так что если `width` и `height` установлены в 100, то для представления 10 000 пикселей холст выделит 40 000 байтов.

Атрибуты `width` и `height` также задают стандартный размер (в пикселях CSS), с которым холст будет отображаться на экране. Если `window.devicePixelRatio` равно 2, то 100×100 пикселей CSS на самом деле составляет 40 000 аппаратных пикселей. Когда содержимое холста рисуется на экране, то 10 000 пикселей в памяти понадобится увеличить, чтобы охватить 40 000 физических пикселей на экране, поэтому ваша графика не будет настолько четкой, как могла бы быть.

Чтобы обеспечить оптимальное качество изображений, не следует использовать атрибуты `width` и `height` для установки экранного размера холста. Взамен необходимо устанавливать желательный экранный размер холста в пикселях CSS с помощью атрибутов `width` и `height` стилей CSS. Затем перед началом рисования в коде JavaScript нужно установить свойства `width` и `height` объекта холста в количество пикселей CSS, умноженное на `window.devicePixelRatio`. Продолжая предыдущий пример, такая методика приведет к отображению холста размером 100×100 пикселей CSS, но с выделением памяти для 200×200 пикселей. (Даже при подобном подходе пользователь может увеличивать масштаб холста и видеть нечеткую или пикселизированную графику, в противополож-

ность графике SVG, которая остается четкой независимо от экранного размера или степени масштабирования.)

15.8.3. Графические атрибуты

В примере 15.5 устанавливались свойства `fillStyle`, `strokeStyle` и `lineWidth` объекта контекста холста. Эти свойства являются графическими атрибутами, которые указывают цвет, применяемый методами `fill()` и `stroke()`, и ширину линий, вычерчиваемых методом `stroke()`. Обратите внимание, что такие параметры не передаются методам `fill()` и `stroke()`, а взамен входят в состав общего *графического состояния* холста. Если вы определяете метод, рисующий фигуру, и не устанавливаете указанные свойства самостоятельно, тогда цвет фигуры можно определить в коде перед вызовом вашего метода, устанавливая свойства `strokeStyle` и `fillStyle`. Такое отделение графического состояния от команд рисования фундаментально для API-интерфейса Canvas и похоже на отделение представления от содержимого, достигаемое за счет применения таблиц стилей CSS к HTML-документам. У объекта контекста есть ряд свойств (и несколько методов), которые влияют на графическое состояние холста. Они рассматриваются ниже.

Стили линий

Свойство `lineWidth` указывает, насколько толстой (в пикселях CSS) будут линии, рисуемые методом `stroke()`. Стандартное значение равно 1. Важно понимать, что ширина линии определяется свойством `lineWidth` во время вызова `stroke()`, а не `lineTo()` и других методов построения пути. Для полного понимания свойства `lineWidth` путь следует мысленно представлять себе как бесконечно тонкую одномерную линию. Линии и кривые, вычерчиваемые методом `stroke()`, центрируются относительно пути, с половиной `lineWidth` с каждой стороны. Если вы вычерчиваете замкнутый путь и хотите, чтобы линия отображалась только вне контура, то вычертите сначала путь и затем заполните контур непрозрачным цветом для сокрытия части линии внутри контура. Или если вы хотите, чтобы линия была видна только внутри замкнутого пути, тогда вызовите сначала методы `save()` и `clip()`, а затем `stroke()` и `restore()`. (Методы `save()`, `restore()` и `clip()` будут описаны позже.)

При вычерчивании линий с шириной более двух пикселей свойства `lineCap` и `lineJoin` могут оказывать значительное влияние на внешний вид концов пути и вершин, в которых встречаются два отрезка пути. На рис. 15.9 показаны значения и результирующее графическое представление `lineCap` и `lineJoin`.

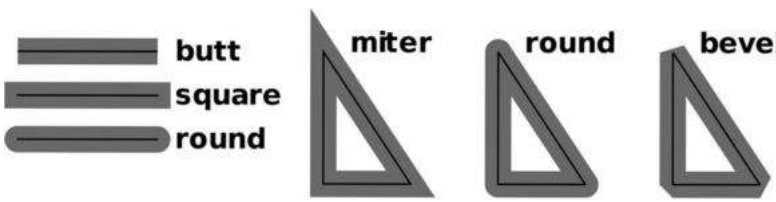


Рис. 15.9. Атрибуты `lineCap` и `lineJoin`

Стандартным значением lineCap является "butt", а стандартным значением lineJoin — "miter". Тем не менее, обратите внимание, что если две линии встречаются под очень малым углом, тогда результирующий конус может стать очень длинным и визуально непривлекательным. Если конус в данной вершине будет длиннее половины ширины линии, умноженной на значение свойства miterLimit, то вершина будет рисоваться со скошенным, а не конусным соединением. Стандартное значение miterLimit равно 10.

Метод stroke() способен вычерчивать пунктирные и точечные, а также сплошные линии, и графическое состояние холста включает массив чисел, который служит "шаблоном штриховки", указывая количество рисуемых и количество пропускаемых точек. В отличие от других свойств вычерчивания линий шаблон штриховки устанавливается и запрашивается посредством методов setLineDash() и getLineDash(), а не с помощью свойства. Чтобы указать точечный шаблон штриховки, метод setLineDash() можно вызвать так:

```
c.setLineDash([18, 3, 3, 3]); // Штрих 18 пикселей, пропуск 3 пикселя,
                              // точка 3 пикселя, пропуск 3 пикселя.
```

Наконец, свойство lineDashOffset указывает, с какого места должно начинаться вычерчивание шаблона штриховки. Стандартное значение lineDashOffset равно 0. Пути, вычерченные с использованием показанного здесь шаблона штриховки, начинаются со штриха 18 пикселей, но если lineDashOffset установлено в 21, тогда тот же самый путь будет начинаться с точки, за которой следует пропуск и штрих.

Цвета, шаблоны и градиенты

Свойства fillStyle и strokeStyle определяют, как пути заполняются и вычерчиваются. Слово Style часто означает цвет, но данные свойства также можно применять для указания цветового градиента или изображения, используемого для заполнения и вычерчивания. (Обратите внимание, что рисование линии по существу является тем же, что и заполнение узкой области между двумя сторонами линии, а потому заполнение и вычерчивание — фундаментально одна и та же операция.)

Если вы хотите заполнять или вычерчивать сплошным цветом (или полупрозрачным цветом), тогда просто установите эти свойства в допустимую строку цвета CSS. Ничего другого не потребуется.

Чтобы заполнять (или чертить) с помощью цветового градиента, установите fillStyle (или strokeStyle) в объект CanvasGradient, возвращаемый методом createLinearGradient() или createRadialGradient() объекта контекста. В аргументах метода createLinearGradient() указываются координаты двух точек, которые определяют линию (она не обязана быть горизонтальной или вертикальной), вдоль которой будут варьироваться цвета. В аргументах метода createRadialGradient() указываются центры и радиусы двух кругов. (Они не обязаны быть концентрическими, но первый круг обычно находится полностью внутри второго.) Области внутри меньшего круга или вне большего круга будут заполняться сплошными цветами, а области между двумя кругами — цветовым градиентом.

После создания объекта CanvasGradient, определяющего области холста, которые должны быть заполнены, потребуется определить цвета градиента, вызывая метод addColorStop() объекта CanvasGradient. В первом аргументе данного метода передается число между 0.0 и 1.0, а во втором — спецификация цвета CSS. Вы должны вызвать метод addColorStop() хотя бы дважды, чтобы определить простой цветовой градиент, но можете вызывать его больше двух раз. Цвет в позиции 0.0 будет появляться в начале градиента, а цвет в позиции 1.0 — в конце. Если вы указываете дополнительные цвета, тогда они будут появляться в заданных дробных позициях внутри градиента. Между указанными позициями цвета будут плавно интерполироваться. Ниже приведены примеры:

```
// Линейный градиент по диагонали холста
// (предполагается отсутствие трансформаций).
let bgfade = c.createLinearGradient(0,0,canvas.width,canvas.height);
bgfade.addColorStop(0.0, "#88f");   // Начать со светло-голубого
                                    // в левом верхнем углу.
bgfade.addColorStop(1.0, "#fff");   // Постепенно изменять до белого
                                    // в нижнем правом углу.

// Градиент между двумя концентрическими кругами.
// Прозрачный в середине, плавно переходящий в полупрозрачный серый
// и затем снова в прозрачный.
let donut = c.createRadialGradient(300,300,100, 300,300,300);
donut.addColorStop(0.0, "transparent");           // Прозрачный
donut.addColorStop(0.7, "rgba(100,100,100,.9)"); // Полупрозрачный
                                                  // серый
donut.addColorStop(1.0, "rgba(0,0,0,0)");         // Снова прозрачный
```

Важно понимать, что градиенты не являются независимыми от позиции. Когда вы создаете градиент, то указываете для него границы. Если затем вы попытаетесь заполнить область за пределами этих границ, то получите сплошной цвет на одном или другом конце градиента.

В дополнение к цветам и цветовым градиентам вы также можете заполнять и вычерчивать с применением изображений, для чего необходимо установить fillStyle или strokeStyle в объект CanvasPattern, возвращаемый методом createPattern() объекта контекста. Первым аргументом метода createPattern() должен быть элемент <img> или <canvas>, содержащий изображение, с использованием которого вы хотите заполнять или чертить. (Обратите внимание, что для применения подобным способом исходное изображение или холст вставлять в документ не обязательно.) Во втором аргументе метода createPattern() передается строка "repeat", "repeat-x", "repeat-y" или "no-repeat", которая указывает, должно ли фоновое изображение повторяться (и в каких направлениях).

Стили текста

Свойство font указывает шрифт для использования методами рисования текста fillText() и strokeText() (см. подраздел "Текст" далее в главе). Значением свойства font должна быть строка с таким же синтаксисом, как в CSS-атрибуте font.

Свойство textAlign указывает, каким образом текст должен выравниваться по горизонтали относительно координаты x, переданной fillText() или strokeText(). Допустимыми значениями являются "start", "left", "center", "right" и "end". По умолчанию принимается значение "start", которое для текста, записываемого слева направо, имеет такой же смысл, как "left".

Свойство textBaseline указывает, каким образом текст должен выравниваться по вертикали относительно координаты y. По умолчанию принимается значение "alphabetic", которое подходит для Latin и похожих систем письма. Значение "ideographic" предназначено для применения с такими системами письма, как китайская и японская. Значение "hanging" предназначено для использования с деванагари и похожими системами письма (которые применяются для многих языков в Индии). Базовые линии "top", "middle" и "bottom" — это чисто геометрические базовые линии, основанные на "кегельной площадке" (em square) шрифта.

Тени

Четыре свойства объекта контекста управляют рисованием падающих теней. Если вы надлежащим образом установите такие свойства, тогда любая вычерчиваемая линия, область, текст или изображение будет иметь тень, которая создаст впечатление, что указанный элемент парит над поверхностью холста.

Свойство shadowColor задает цвет тени. По умолчанию это полностью прозрачный черный цвет, и тени не появятся о тех пор, пока вы не установите shadowColor в полупрозрачный или непрозрачный цвет. Свойство shadowColor может быть установлено только в строку цвета: шаблоны и градиенты для теней не разрешены. Использование полупрозрачного цвета для теней обеспечивает наиболее реалистичные эффекты тени, т.к. через тень просматривается фон.

Свойства shadowOffsetX и shadowOffsetY указывают смещения тени по осям x и y. По умолчанию оба свойства принимаются равными 0, что приводит к размещению тени прямо под рисунком, где она не видна. Если вы установите оба свойства в положительное значение, то тени будут появляться ниже и правее рисунка, как будто сверху слева находится источник света, освещающих холст извне экрана компьютера. Более крупные значения смещений производят тени большего размера и делают вычерченные объекты выглядящими так, как если бы они парили "выше" над холстом. Такие значения не влияют на трансформации координат (см. подраздел 15.8.5): направление тени и "высота" остаются согласованными, даже когда фигуры вращаются и масштабируются.

Свойство shadowBlur указывает, насколько размыты края тени. Стандартное значение равно 0, что дает четкие тени без размытия. Большие значения создают больше размытости вплоть до верхней границы, определяемой реализацией.

Полупрозрачность и наложение

Если вы хотите вычертить или заполнить путь с применением полупрозрачного цвета, тогда можете установить свойство strokeStyle или fillStyle,

используя синтаксис цветов CSS вроде `rgba(...)`, который поддерживает альфа-прозрачность. Буква "A" в "RGBA" означает "alpha" (альфа) и представляет собой значение между 0 (полная прозрачность) и 1 (полная непрозрачность). Но API-интерфейс Canvas предлагает еще один способ работы с полупрозрачными цветами. Если вы не хотите явно указывать альфа-канал для каждого цвета либо желаете добавить полупрозрачность к непрозрачным изображениям или шаблонам, то можете установить свойство `globalAlpha`. Каждый рисуемый вами пиксель будет иметь свое альфа-значение, умноженное на `globalAlpha`. По умолчанию значение `globalAlpha` принимается равным 1, что не добавляет никакой прозрачности. Если вы установите `globalAlpha` в 0, тогда все, что вы будете вычерчивать, окажется прозрачным, и на холсте ничего не появится. Но если вы установите `globalAlpha` в `0.5`, то пиксели, которые иначе были бы непрозрачными, станут непрозрачными на 50%, а пиксели, которые были бы непрозрачными на 50%, станут непрозрачными на 25%.

Когда вы вычерчиваете линии, заполняете области, рисуете текст или копируете изображения, то обычно ожидаете, что новые пиксели будут появляться поверх пикселей, которые уже находятся на холсте. Если вы рисуете посредством непрозрачных пикселей, то они просто замещают пиксели, которые уже там были. Если вы рисуете с помощью полупрозрачных пикселей, тогда новый ("исходный") пиксель объединяется со старым ("целевым") пикселем, так что старый пиксель проглядывает сквозь новый пиксель в зависимости от того, насколько он прозрачен.

Такой процесс объединения новых (возможно полупрозрачных) исходных пикселей с существующими (возможно полупрозрачными) целевыми пикселями называется *наложением*, и описанный ранее процесс наложения является стандартным способом, которым API-интерфейс Canvas объединяет пиксели. Но вы можете установить свойство `globalCompositeOperation`, чтобы определить другие способы объединения пикселей. По умолчанию принимается значение `"source-over"`, которое приводит к тому, что исходные пиксели рисуются "поверх" целевых пикселей и объединяются с ними, если исходные пиксели полупрозрачные. В случае установки `globalCompositeOperation` в `"destination-over"` холст будет объединять пиксели, как если бы новые исходные пиксели были нарисованы под существующими целевыми пикселями. Если целевой пиксель полупрозрачен или прозрачен, тогда некоторая доля или весь цвет исходного пикселя будет виден в результирующем цвете. В качестве еще одного примера: в режиме наложения `"source-atop"` исходные пиксели так объединяются с прозрачностью целевых пикселей, что на частях холста, которые уже полностью прозрачны, ничего не рисуется. Для свойства `globalCompositeOperation` существует несколько допустимых значений, но большинство из них имеют только специализированные применения и здесь не рассматриваются.

Сохранение и восстановление графического состояния

Поскольку API-интерфейс Canvas определяет графические атрибуты в объекте контекста, у вас может возникнуть соблазн вызывать `getContext()` много

раз, чтобы получить множество объектов контекста. Если бы это было возможно, тогда вы могли бы определять для каждого контекста разные атрибуты: затем каждый контекст был бы похож на отдельную кисть и рисовал своим цветом или вычерчивал линии отличающейся ширины. К сожалению, использовать холст подобным образом нельзя. Каждый элемент <canvas> имеет только один объект контекста, и любой вызов `getContext()` возвращает тот же самый объект `CanvasRenderingContext2D`.

Хотя API-интерфейс Canvas разрешает определять только одиночный набор графических атрибутов за раз, он позволяет сохранять текущее графическое состояние, так что вы можете изменить его и позже легко восстановить. Метод `save()` помещает текущее графическое состояние в стек сохраненных состояний. Метод `restore()` извлекает из стека и восстанавливает самое последнее сохраненное состояние. Все свойства, которые были описаны в настоящем разделе, являются частью сохраненного состояния, как и текущая трансформация и область отсечения (будут объясняться далее). Важно отметить, что определенный в текущий момент путь и текущая точка в сохраненное состояние не входят и не могут быть сохранены и восстановлены.

15.8.4. Операции рисования холста

Вы уже видели ряд базовых методов холста — `beginPath()`, `moveTo()`, `lineTo()`, `closePath()`, `fill()` и `stroke()` — для определения, заполнения и рисования линий и многоугольников. Но в состав API-интерфейса Canvas входят также и другие методы рисования.

Прямоугольники

В объекте `CanvasRenderingContext2D` определены четыре метода для рисования прямоугольников. Все четыре ожидают двух аргументов, которые описывают один угол прямоугольника, а за ними следуют ширина и высота прямоугольника. Обычно указывается левый верхний угол и передаются положительные ширина и высота, но можно также указывать другие углы и передавать отрицательные размеры.

Метод `fillRect()` заполняет указанный прямоугольник с применением текущего стиля `fillStyle`. Метод `strokeRect()` вычерчивает контур указанного прямоугольника, используя текущий стиль `strokeStyle` и другие атрибуты линий. Метод `clearRect()` похож на `fillRect()`, но игнорирует текущий стиль заполнения и заполняет прямоугольник с применением прозрачных черных пикселей (стандартный цвет для всех пустых холстов). В этих трех методах важно то, что они не влияют на текущий путь или текущую точку внутри данного пути.

Последний метод для прямоугольников называется `rect()` и влияет на текущий путь: он добавляет к пути указанный прямоугольник в собственном подпути. Подобно другим методам определения путей сам он ничего не заполняет и не вычерчивает.

Кривые

Путь представляет собой последовательность подпутей, а подпуть — последовательность соединенных точек. В путях, которые были определены в подразделе 15.8.1, точки соединялись отрезками прямой линии, но так нужно не всегда. В объекте `CanvasRenderingContext2D` определено несколько методов, которые добавляют новую точку в подпуть и соединяют текущую точку с новой точкой, используя кривую.

- **`arc()`**. Этот метод добавляет к пути окружность или часть окружности (дугу). Рисуемая дуга указывается с помощью шести параметров: координаты *x* и *y* центра окружности, радиус окружности, начальный и конечный углы дуги и направление (по часовой стрелке или против часовой стрелки) дуги между заданными двумя углами. Если в пути имеется текущая точка, тогда метод соединяет ее с началом дуги посредством прямой линии (что полезно при рисовании клиньев или кусков пирога) и затем соединяет начало дуги с концом дуги с применением части окружности, делая конец дуги новой текущей точкой. Если текущей точки нет, когда метод вызван, тогда он только добавляет дугу окружности к пути.
- **`ellipse()`**. Этот метод во многом похож на `arc()`, но он добавляет к пути эллипс или его часть. Вместо одного радиуса указываются два: радиус по оси *x* и радиус по оси *y*. Кроме того, поскольку эллипсы не являются радиально симметричными, метод принимает еще один аргумент, где указывается число радиан, на которые эллипс поворачивается по часовой стрелке относительно своего центра.
- **`arcTo()`**. Этот метод рисует прямую линию и дугу окружности, как делает метод `arc()`, но дуга указывается с использованием других параметров. Аргументы `arcTo()` определяют точки P1 и P2 и радиус. Добавляемая к пути дуга имеет указанный радиус. Она начинается в точке касания с (воображаемой) линией из текущей точки до P1 и заканчивается в точке касания с (воображаемой) линией между P1 и P2. Такой необычно выглядящий метод указания дуг на самом деле очень удобен при рисовании фигур со скругленными углами. В случае указания радиуса 0 метод просто рисует прямую линию из текущей точки в P1. Однако для ненулевого радиуса он рисует прямую линию из текущей точки в направлении P1 и затем изгибает эту линию по окружности, пока она не станет ориентированной в направлении P2.
- **`bezierCurveTo()`**. Этот метод добавляет новую точку P в подпуть и соединяет ее с текущей точкой посредством кубической кривой Безье. Форма кривой указывается двумя "управляющими точками" C1 и C2. В начале кривой (в текущей точке) кривая ориентирована в направлении C1. В конце кривой (в точке P) кривая приходит со стороны C2. Между этими точками направление кривой плавно варьируется. Точка P становится новой текущей точкой подпути.
- **`quadraticCurveTo()`**. Этот метод похож на `bezierCurveTo()`, но применяет квадратичную кривую Безье, а не кубическую, и имеет только одну управляющую точку.

Перечисленные методы можно использовать для вычерчивания путей вроде показанных на рис. 15.10.

Рис. 15.10. Криволинейные пути на холсте

В примере 15.6 приведен код, который применялся для получения результата, показанного на рис. 15.10. Методы, продемонстрированные в коде, относятся к наиболее сложным в API-интерфейсе Canvas; полные описания методов и их аргументов ищите в онлайновом справочнике.

Пример 15.6. Добавление кривых к пути

```
// Служебная функция для преобразования углов из градусов в радианы.
function rads(x) { return Math.PI*x/180; }

// Получить объект контекста элемента canvas документа.
let c = document.querySelector("canvas").getContext("2d");

// Определить ряд графических атрибутов и нарисовать кривые.
c.fillStyle = "#aaa";       // Заполнение серым.
c.lineWidth = 2;            // 2-пиксельные черные (по умолчанию) линии.

// Нарисовать окружность.
// Текущей точки нет, поэтому рисуется только окружность
// без прямой линии от текущей точки до начала окружности.
c.beginPath();
c.arc(75,100,50,            // Центр в (75,100), радиус 50.
    0,rads(360),false);     // Двигаться по часовой стрелке от 0 до 360 градусов
c.fill();                   // Заполнить окружность.
c.stroke();                 // Вычертить ее контур.

// Нарисовать эллипс аналогичным образом.
c.beginPath();              // Начать новый путь, не соединенный с окружностью
c.ellipse(200, 100, 50, 35, rads(15),  // Центр, радиусы и поворот.
   0, rads(360), false);    // Начальный угол, конечный угол, направление.

// Нарисовать клин. Углы измеряются по часовой стрелке относительно
// положительной оси x.
// Обратите внимание, что arc() добавляет линию из текущей точки до начала дуги.
c.moveTo(325, 100);         // Начать в центре окружности.
c.arc(325, 100, 50,         // Центр и радиус окружности.
    rads(-60), rads(0),     // Начать с угла -60 и двигаться до угла 0
    true);                  // против часовой стрелки.
c.closePath();              // Добавить радиус обратно к центру окружности.

// Нарисовать похожий клин, чуть смещенный и в противоположном направлении.
c.moveTo(340, 92);
c.arc(340, 92, 42, rads(-60), rads(0), false);
c.closePath();
```

```
// Использовать arcTo() для скругленных углов. Здесь рисуется квадрат
// с левым верхним углом в (400,50) и скругленными углами с разными радиусами.
c.moveTo(450, 50);                  // Начать в середине верхнего ребра.
c.arcTo(500,50,500,150,30);         // Добавить часть верхнего ребра
                                    // и правый верхний угол.
c.arcTo(500,150,400,150,20);        // Добавить правое ребро и правый нижний угол.
c.arcTo(400,150,400,50,10);         // Добавить нижнее ребро и левый нижний угол.
c.arcTo(400,50,500,50,0);           // Добавить левое ребро и левый верхний угол.
c.closePath();                      // Замкнуть путь и добавить остаток
                                    // верхнего ребра.

// Квадратичная кривая Безье: одна управляющая точка.
c.moveTo(525, 125);                 // Начать здесь.
c.quadraticCurveTo(550, 75, 625, 125);    // Рисовать кривую до (625, 125).
c.fillRect(550-3, 75-3, 6, 6);  // Пометить управляющую точку (550,75).

// Кубическая кривая Безье.
c.moveTo(625, 100);                 // Начать в (625, 100).
c.bezierCurveTo(645,70,705,130,725,100); // Рисовать кривую до (725, 100).
c.fillRect(645-3, 70-3, 6, 6);  // Пометить управляющие точки.
c.fillRect(705-3, 130-3, 6, 6);

// В заключение заполнить кривые и вычертить их контуры.
c.fill();
c.stroke();
```

Текст

Для рисования текста на холсте обычно используется метод `fillText()`, который рисует текст с применением цвета (либо градиента или шаблона), указанного в свойстве `fillStyle`. Для обеспечения специальных эффектов в тексте с большими размерами можно использовать `strokeText()`, чтобы рисовать контуры индивидуальных глифов шрифта. Оба метода принимают в своем первом аргументе текст, подлежащий рисованию, а во втором и третьем аргументах — координаты x и y текста. Ни тот, ни другой метод не влияют на текущий путь или текущую точку.

Методы `fillText()` и `strokeText()` принимают необязательный четвертый аргумент, который в случае предоставления указывает максимальную ширину отображаемого текста. Если текст будет шире указанного значения, когда он нарисован с применением свойства `font`, то холст подгонит его за счет масштабирования либо использования более узкого или меньшего шрифта.

Если вам необходимо самостоятельно измерить текст до его рисования, тогда передайте его методу `measureText()`. Этот метод возвращает объект `TextMetrics`, который указывает измерения текста, когда он нарисован с применением текущего свойства `font`. На момент написания главы единственной "метрикой", содержащейся в объекте `TextMetrics`, была ширина. Экранная ширина запрашивается с помощью строки следующего вида:

```
let width = c.measureText(text).width;
```

Поступать так полезно, например, если вы хотите центрировать строку текста внутри холста.

Изображения

В дополнение к векторной графике (пути, линии и т.д.) в API-интерфейсе Canvas также поддерживаются растровые изображения. Метод `drawImage()` копирует пиксели исходного изображения (или прямоугольной области внутри исходного изображения) на холст, при необходимости масштабируя и поворачивая пиксели изображения. Метод `drawImage()` можно вызывать с тремя, пятью или девятью аргументами. Во всех случаях первым аргументом является исходное изображение, из которого должны копироваться пиксели. Аргументом изображения часто выступает элемент `<img>`, но им также может быть еще один элемент `<canvas>` или даже элемент `<video>` (из которого будет копироваться одиночный кадр). Если вы укажете элемент `<img>` или `<video>`, который не закончил загружать свои данные, тогда вызов `drawImage()` ничего не будет делать.

В версии `drawImage()` с тремя аргументами второй и третий аргументы задают координаты x и y, где должен рисоваться левый верхний угол изображения. В этой версии метода на холст копируется целое исходное изображение. Координаты x и y интерпретируются в текущей системе координат, а изображение при необходимости масштабируется и поворачивается в зависимости от того, какая трансформация холста активна в текущий момент.

Версия `drawImage()` с пятью аргументами добавляет к описанным ранее аргументам x и y аргументы `width` и `height`. Указанные четыре аргумента определяют прямоугольник назначения внутри холста. Левый верхний угол исходного изображения попадает в `(x,y)`, а правый нижний угол — в `(x+width, y+height)`. Копируется все исходное изображение. В данной версии метода исходное изображение будет масштабироваться, чтобы уместиться в прямоугольнике назначения.

В версии `drawImage()` с девятью аргументами указываются исходный прямоугольник и прямоугольник назначения и копируются только пиксели внутри исходного прямоугольника. Аргументы со второго по пятый задают исходный прямоугольник. Они измеряются в пикселях CSS. Если исходное изображение является еще одним холстом, то исходный прямоугольник использует стандартную систему координат для этого холста и игнорирует любые трансформации, которые были указаны. Аргументы с шестого по девятый определяют прямоугольник назначения, в котором рисуется изображение, и находятся в текущей системе координат холста, а не в стандартной системе координат.

Помимо рисования изображений на холсте мы также можем извлекать содержимое холста как изображение с применением метода `toDataURL()`. В отличие от всех остальных методов, которые здесь описаны, `toDataURL()` представляет собой метод самого элемента холста, а не объекта контекста. Обычно метод `toDataURL()` вызывается без аргументов и возвращает содержимое холста в виде изображения PNG, закодированного в форме строки с использованием URL типа `data:`. Возвращенный URL подходит для применения с элементом `<img>` и вот как можно сделать статический снимок холста:

```
let img = document.createElement("img"); // Создать элемент <img>.
img.src = canvas.toDataURL();            // Установить его атрибут src.
document.body.appendChild(img);          // Добавить его в документ.
```

15.8.5. Трансформации системы координат

Как уже отмечалось, в стандартной системе координат холста начало находится в левом верхнем углу, координаты *x* увеличиваются вправо, а координаты *y* увеличиваются вниз. В стандартной системе координаты точки отображаются прямо на пиксель CSS (который затем отображается непосредственно на один или большее количество аппаратных пикселей). Определенные операции и атрибуты холста (такие как извлечение низкоуровневых значений пикселей и установка смещений тени) всегда используют такую стандартную систему координат. Тем не менее, помимо стандартной системы координат каждый холст имеет "текущую матрицу преобразования" как часть графического состояния. Такая матрица определяет текущую систему координат холста. В большинстве операций холста, когда указаны координаты какой-то точки, она считается точкой в текущей системе координат, а не в стандартной системе координат. Текущая матрица преобразования применяется для того, чтобы преобразовывать указываемые координаты в эквивалентные координаты в стандартной системе координат.

Метод `setTransform()` позволяет устанавливать матрицу преобразования холста напрямую, но трансформации системы координат обычно легче указывать как последовательность операций перемещения, поворота и масштабирования. На рис. 15.11 иллюстрируются такие операции и их влияние на систему координат холста. Программа, которая выдала показанный результат, семь раз подряд рисовала тот же самый набор осей. Единственное, что каждый раз менялось — текущая трансформация. Обратите внимание, что трансформации влияют как на текст, так и на нарисованные линии.

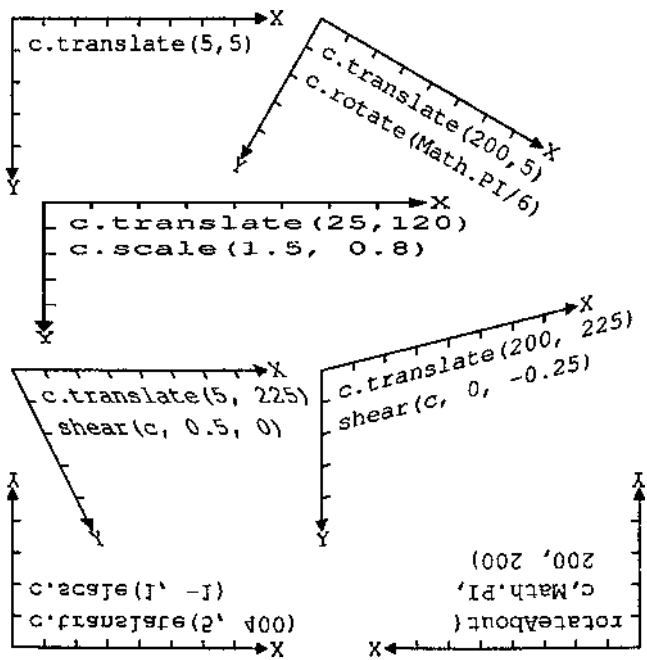


***Рис. 15.11.** Примеры трансформаций системы координат*

Метод `translate()` просто перемещает начало системы координат влево, вправо, вверх или вниз. Метод `rotate()` поворачивает оси по часовой стрелке на указанный угол. (В API-интерфейсе Canvas углы всегда задаются в радианах. Чтобы преобразовать градусы в радианы, поделите их на 180 и умножьте на `Math.PI`.) Метод `scale()` растягивает или сжимает расстояния вдоль оси *x* или *y*.

Передача методу `scale()` отрицательного масштабного коэффициента приводит к переворачиванию оси относительно начала координат, как если бы она отражалась в зеркале. Именно это и делалось слева внизу на рис. 15.11: метод `translate()` использовался для перемещения начала координат в левый нижний угол холста, после чего с применением метода `scale()` ось *y* была зеркально отображена, чтобы координаты *y* увеличивались при движении к верху страницы. Такая перевернутая система координат должна быть знакома из курса алгебры и может оказаться полезной для вычерчивания точек данных на диаграммах. Однако обратите внимание, что в итоге текст становится трудным для чтения!

Математическое восприятие трансформаций

Я считаю, что трансформации легче всего воспринимать с точки зрения геометрии, думая о методах `translate()`, `rotate()` и `scale()` как о преобразовании осей системы координат, что было показано на рис. 15.11. Кроме того, трансформации также можно воспринимать с точки зрения алгебры как уравнения, которые отображают координаты точки `(x,y)` в трансформированной системе координат обратно на координаты `(x',y')` той же точки в предыдущей системе координат.

Вызов метода `c.translate(dx,dy)` можно описать с помощью приведенных ниже уравнений:

```
x' = x + dx; //Координата X нуля в новой системе - это dx в старой системе
y' = y + dy;
```

Операции масштабирования имеют такие же простые уравнения. Вот как можно описать вызов `c.scale(sx,sy)`:

```
x' = sx * x;
y' = sy * y;
```

С поворотами дело обстоит сложнее. Вызов `c.rotate(a)` описывается следующими тригонометрическими уравнениями:

```
x' = x * cos(a) - y * sin(a);
y' = y * cos(a) + x * sin(a);
```

Имейте в виду, что порядок преобразований имеет значение. Предположим, что мы начинаем со стандартной системы координат холста, затем перемещаем ее и далее масштабируем. Чтобы отобразить точку `(x,y)` в текущей системе координат обратно на точку `(x'',y'')` в стандартной системе координат, мы должны сначала применить уравнения масштабирования для отображения точки на промежуточную точку `(x',y')` в перемещенной, но не масштабированной системе координат, после чего использовать уравнения масштабирования для отображения промежуточной точки на точку `(x'',y'')`.

Вот результат:

```
x'' = sx*x + dx;
y'' = sy*y + dy;
```

С другой стороны, если мы вызовем `scale()` до вызова `translate()`, то результирующие уравнения будут другими:

```
x'' = sx*(x + dx);
y'' = sy*(y + dy);
```

При восприятии последовательностей трансформаций с точки зрения алгебры важно помнить о том, что мы обязаны работать в обратном направлении от последней (самой недавней) трансформации к первой. Тем не менее, при восприятии трансформированных осей с точки зрения геометрии мы работаем в прямом направлении от первой трансформации к последней.

Трансформации, поддерживаемые холстом, известны как аффинные преобразования. Аффинные преобразования могут изменять расстояния между точками и углы между линиями, но после аффинного преобразования параллельные линии остаются параллельными — например, задать искажение в стиле линзы "рыбий глаз" с аффинным преобразованием невозможно. Произвольное аффинное преобразование может быть описано шестью параметрами от `a` до `f` в следующих уравнениях:

```
x' = ax + cy + e
y' = bx + dy + f
```

Вы можете применить произвольное преобразование к текущей системе координат, передавая эти шесть параметров в метод `transform()`. На рис. 15.11 иллюстрировались два типа преобразований — сдвиги и повороты, которые вы можете реализовать с помощью метода `transform()`:

```
// Трансформация типа сдвига:
// x' = x + kx*y;
// y' = ky*x + y;
function shear(c, kx, ky) { c.transform(1, ky, kx, 1, 0, 0); }

// Поворачивает на theta радиан против часовой стрелки
// относительно точки (x,y).
// Этого также можно достичь с помощью последовательности
// из перемещения, поворота, перемещения.
function rotateAbout(c, theta, x, y) {
    let ct = Math.cos(theta);
    let st = Math.sin(theta);
    c.transform(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);
}
```

Метод `setTransform()` принимает такие же аргументы, как `transform()`, но вместо трансформирования текущей системы координат он игнорирует текущую систему, трансформирует стандартную систему координат и делает результат новой текущей системой координат. Метод `setTransform()` удобен для временного сброса холста в его стандартную систему координат:

```
c.save();                          // Сохранить текущую систему координат.
c.setTransform(1,0,0,1,0,0);       // Возвратиться к стандартной системе
                                   // координат.
//Выполнить операции, используя стандартные координаты в пикселях CSS.
c.restore();          // Восстановить сохраненную систему координат.
```

Пример трансформации

В примере 15.7 демонстрируется мощь трансформаций системы координат за счет рекурсивного использования методов `translate()`, `rotate()` и `scale()` для рисования фрактальной кривой Коха (т.н. снежинки Коха). Вывод примера показан на рис. 15.12, где представлены кривые Коха с уровнями рекурсии 0, 1, 2, 3 и 4.

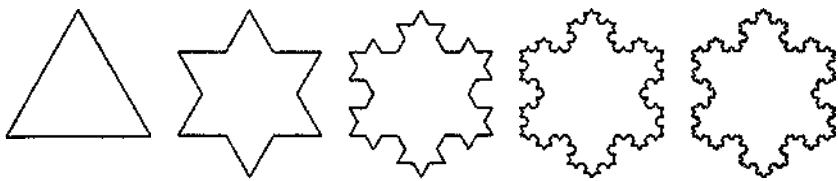


Рис. 15.12. Кривые Коха

Код, производящий такие фигуры, весьма элегантен, но из-за применения в нем рекурсивных трансформаций системы координат понять его не особенно легко. Даже не учитывая все нюансы, обратите внимание, что код включает только один вызов метода `lineTo()`. Каждый одиночный отрезок прямой на рис. 15.12 рисуется примерно так:

```
c.lineTo(len, 0);
```

Значение переменной `len` не изменяется во время выполнения программы, поэтому позиция, ориентация и длина каждого отрезка прямой определяется операциями перемещения, поворота и масштабирования.

Пример 15.7. Рисование кривых Коха с помощью трансформаций

```
let deg = Math.PI/180;   // Для преобразования градусов в радианы.

// Рисует фрактальную кривую Коха уровня n на холсте c
// с левым нижним углом в (x,y) и длиной стороны len.
function snowflake(c, n, x, y, len) {
    c.save();              // Сохранить текущую трансформацию.
    c.translate(x,y);      // Переместить начало координат в стартовую точку.
    c.moveTo(0,0);         // Начать новый подпуть в новом начале координат.
    leg(n);                // Нарисовать первый луч кривой.
    c.rotate(-120*deg);// Повернуть на 120 градусов против часовой стрелки.
    leg(n);                // Нарисовать второй луч.
    c.rotate(-120*deg);// Снова повернуть.
    leg(n);                // Нарисовать финальный луч.
    c.closePath();         // Закрыть подпуть.
    c.restore();           // Восстановить первоначальную трансформацию.

    // Рисует одиночный луч кривой Коха.
```

```
    // Эта функция оставляет текущую точку в конце нарисованного луча
    // и перемещает систему координат, чтобы текущей точкой стала (0,0).
    // Таким образом, мы можем легко вызвать rotate() после рисования луча.
    function leg(n) {
        c.save();              // Сохранить текущую трансформацию.
        if (n === 0) {         // Нерекурсивный случай:
          c.lineTo(len, 0);    // просто нарисовать горизонтальную линию.
        }                      //
        else {       // Рекурсивный случай: нарисовать 4 подлуча наподобие \/
          c.scale(1/3,1/3);    // Подлучи имеют 1/3 размера этого луча.
          leg(n-1);            // Рекурсивный вызов для первого подлуча.
          c.rotate(60*deg);    // Повернуть на 60 градусов по часовой стрелке.
          leg(n-1);            // Второй подлуч.
          c.rotate(-120*deg);// Повернуть на 120 градусов обратно.
          leg(n-1);            // Третий подлуч.
          c.rotate(60*deg);    // Повернуть в первоначальном направлении.
          leg(n-1);            // Финальный подлуч.
        }
        c.restore();           // Восстановить трансформацию.
        c.translate(len, 0); // Но переместить, чтобы сделать конец луча (0,0)
    }
}
let c = document.querySelector("canvas").getContext("2d");
snowflake(c, 0, 25, 125, 125);  // Кривая уровня 0 является треугольником.
snowflake(c, 1, 175, 125, 125); //Кривая уровня 1 является шестисторонней звездой
snowflake(c, 2, 325, 125, 125); // И т.д.
snowflake(c, 3, 475, 125, 125);
snowflake(c, 4, 625, 125, 125); // Кривая уровня 4 выглядит как снежинка!
c.stroke();                     // Вычертить этот очень сложный путь.
```

15.8.6. Отсечение

После определения пути вы обычно вызываете метод `stroke()` или `fill()` (или оба). Вы также можете вызвать метод `clip()`, чтобы определить область отсечения. Как только область отсечения определена, за ее пределами ничего рисоваться не будет. На рис. 15.13 показан сложный рисунок, полученный с использованием областей отсечения. Вертикальная полоса посередине, идущая вниз, и текст вдоль низа рисунка были вычерчены без областей отсечения и затем заполнены после определения треугольной области отсечения.

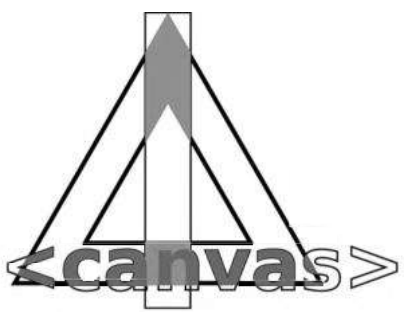


Рис. 15.13. Вычерчивание без отсечения и заполнение с отсечением

Изображение на рис. 15.13 было сгенерировано с применением метода polygon() из примера 15.5 и следующего кода:

```
// Определить ряд атрибутов рисования.
c.font = "bold 60pt sans-serif";     // Большой шрифт.
c.lineWidth = 2;                     // Узкие линии.
c.strokeStyle = "#000";              // Черные линии.

// Контуры прямоугольника и текста.
c.strokeRect(175, 25, 50, 325);      //Вертикальная полоса посередине,
                                     //идущая вниз.
c.strokeText("<canvas>", 15, 330);   // Обратите внимание на вызов
                                     // strokeText(), а не fillText().

// Определить сложный путь с внутренней частью, выходящей наружу.
polygon(c,3,200,225,200);            // Крупный треугольник.
polygon(c,3,200,225,100,0,true);     // Меньший обратный треугольник
                                     // внутри.

// Сделать этот путь областью отсечения.
c.clip();

//Вычертить путь с 5-пиксельной линией полностью внутри области отсечения
c.lineWidth = 10; // Половина этой 10-пиксельной линии будет отсечена.
c.stroke();

// Заполнить части прямоугольника и текста, которые находятся
// внутри области отсечения.
c.fillStyle = "#aaa";                // Светло-серый цвет.
c.fillRect(175, 25, 50, 325);        // Заполнить вертикальную полосу.
c.fillStyle = "#888";                // Темно-серый цвет.
c.fillText("<canvas>", 15, 330);     // Заполнить текст.
```

Важно отметить, что когда вы вызываете clip(), то сам текущий путь усекается до области отсечения и затем становится новой областью отсечения. Это означает, что метод clip() может уменьшать область отсечения, но не расширять ее. Не существует метода для сброса области отсечения, так что перед вызовом clip() вы обычно должны вызвать save(), чтобы позже можно было вызвать restore(), восстановив область без отсечения.

15.8.7. Манипулирование пикселями

Метод getImageData() возвращает объект ImageData, который представляет низкоуровневые пиксели (как компоненты R, G, B и A) из прямоугольной области холста. Вы можете создавать пустые объекты ImageData с помощью createImageData(). Пиксели в объекте ImageData записываемы, так что вы можете устанавливать их любым желаемым образом и затем копировать их обратно на холст посредством putImageData().

Такие методы манипулирования пикселями обеспечивают крайне низкоуровневый доступ к холсту. Прямоугольник, который вы передаете getImageData(), находится в стандартной системе координат: его размеры измеряются в пикселях CSS и на него не влияет текущая трансформация. Когда вы вызываете putImageData(), указанная вами позиция тоже измеряется в стандартной сис-

теме координат. Кроме того, метод putImageData() игнорирует все графические атрибуты. Он не выполняет никакого наложения, не умножает пиксели на globalAlpha и не рисует тени.

Методы манипулирования пикселями полезны для реализации обработки изображений. В примере 15.8 показано, как создать эффект размытости при движении или "смазывания" вроде того, что видно на рис. 15.14.

Рис. 15.14. Эффект размытости при движении, созданный за счет обработки изображений

В приведенном ниже коде демонстрируется использование методов getImageData() и putImageData(), а также методика прохода и изменения значений пикселей в объекте ImageData.

Пример 15.8. Эффект размытости при движении с применением ImageData

```
// Смазать пиксели прямоугольника вправо, порождая эффект размытости при
// движении, как если бы объекты перемещались справа налево. n должно иметь
// значение 2 или больше. Более высокие значения дают большее смазывание.
// Прямоугольник указывается в стандартной системе координат.
function smear(c, n, x, y, w, h) {
    // Получить объект ImageData, который представляет
    // прямоугольник пикселей, подлежащих смазыванию.
    let pixels = c.getImageData(x, y, w, h);

    // Это смазывание делается на месте и требует только исходного объекта
    // ImageData. Некоторые алгоритмы обработки изображений требуют
    // дополнительного объекта ImageData для сохранения трансформированных
    // значений пикселей. Если нам необходим выходной буфер, то мы могли бы
    // создать новый объект ImageData с теми же размерами:
    //     let output_pixels = c.createImageData(pixels);

    // Получить размеры сетки пикселей в объекте ImageData.
    let width = pixels.width, height = pixels.height;

    // Это байтовый массив, который хранит низкоуровневые данные пикселей,
    // слева направо и сверху вниз. Каждый пиксель занимает 4
    // последовательных байта в порядке R,G,B,A.
    let data = pixels.data;

    // Каждый пиксель после первого в каждой строке размывается за счет
    // его замены 1/n-ной частью собственного значения плюс
    // m/n-ной частью значения предыдущего пикселя.
    let m = n-1;

    for(let row = 0; row < height; row++) {              // Для каждой строки.
      let i = row*width*4 + 4;          // Смещение второго пикселя в строке.
      for(let col = 1; col < width; col++, i += 4) {  // Для каждого столбца.
        data[i] = (data[i] + data[i-4]*m)/n;      //Красный компонент пикселя
```

```
            data[i+1] = (data[i+1] + data[i-3]*m)/n; //Зеленый компонент пикселя
            data[i+2] = (data[i+2] + data[i-2]*m)/n; //Синий компонент пикселя
            data[i+3] = (data[i+3] + data[i-1]*m)/n; //Альфа-компонент пикселя
        }
    }
    //Скопировать данные смазанного изображения обратно в ту же позицию на холсте
    c.putImageData(pixels, x, y);
}
```

15.9. API-интерфейсы Audio

HTML-дескрипторы <audio> и <video> позволяют легко включать аудио- и видеоклипы в состав веб-страниц. Это сложные элементы с обширными API-интерфейсами и нетривиальными пользовательскими интерфейсами. С помощью методов play() и pause() можно управлять воспроизведением мультимедиа-содержимого. Для управления громкостью звука и скоростью воспроизведения можно устанавливать свойства volume и playbackRate. А устанавливая свойство currentTime, можно переходить к определенному моменту времени внутри мультимедиа-содержимого.

Однако мы не будем здесь рассматривать дескрипторы <audio> и <video> более подробно. В последующих подразделах описаны два способа добавления к веб-страницам звуковых эффектов посредством сценариев.

15.9.1. Конструктор Audio()

Чтобы включить звуковые эффекты в веб-страницы, добавлять дескриптор <audio> к HTML-документу вовсе не обязательно. Вы можете динамически создавать элементы <audio> с помощью нормального метода document.createElement() модели DOM или в качестве сокращения просто использовать конструктор Audio(). Добавлять созданный элемент к документу с целью его воспроизведения не понадобится. Вы можете просто вызвать его метод play():

```
// Заблаговременно загрузить звуковой эффект,
// чтобы он был готов к использованию.
let soundeffect = new Audio("soundeffect.mp3");

// Воспроизводить звуковой эффект всякий раз,
// когда пользователь щелкает кнопкой мыши.
document.addEventListener("click", () => {
    soundeffect.cloneNode().play(); //Загрузить и воспроизвести звук.
});
```

Обратите внимание на применение здесь cloneNode(). Если пользователь быстро щелкает кнопкой мыши, то мы хотим иметь возможность одновременно воспроизводить множество перекрывающихся копий звукового эффекта. Для этого нам нужно множество элементов <audio>. Поскольку элементы <audio> не добавлялись к документу, они будут подвергнуты сборке мусора, когда закончат воспроизведение.

15.9.2. API-интерфейс WebAudio

Помимо воспроизведения записанных звуков с помощью элементов <audio> веб-браузеры также допускают генерацию и воспроизведение синтезированных звуков посредством API-интерфейса WebAudio. Использование API-интерфейса WebAudio похоже на подключение электронного синтезатора старого образца с помощью коммутационных шнуров. При работе с API-интерфейсом WebAudio вы создаете набор объектов AudioNode, которые представляют источники, трансформации и места назначения форм волн, после чего соединяете такие узлы вместе в сеть для выпуска звуков. API-интерфейс WebAudio не особенно сложен, но полное объяснение требует понимания концепций электронной музыки и обработки сигналов, обсуждение которых выходит за рамки настоящей книги.

В приведенном далее коде API-интерфейс WebAudio применяется для синтеза короткого аккорда, который затухает примерно за секунду. В примере демонстрируются основы API-интерфейса WebAudio. Если вам интересно, тогда можете поискать сведения об этом API-интерфейсе в онлайновых справочниках:

```
// Начать с создания объекта audioContext. Браузер Safari по-прежнему
// требует использования webkitAudioContext вместо AudioContext.
let audioContext = new (this.AudioContext||this.webkitAudioContext)();

// Определить базовый звук как комбинацию из трех чистых
// синусоидальных волн.
let notes = [ 293.7, 370.0, 440.0 ]; //Аккорд ре мажор: ре, фа-диез и ля

// Создать узлы генератора для каждой ноты,
// которые мы хотим воспроизводить.
let oscillators = notes.map(note => {
    let o = audioContext.createOscillator();
    o.frequency.value = note;
    return o;
});

// Формировать звук, регулируя его громкость с течением времени.
// Начиная с момента 0, быстро увеличить громкость до полной.
// Затем, начиная с момента 0.1, медленно уменьшить громкость до 0.
let volumeControl = audioContext.createGain();
volumeControl.gain.setTargetAtTime(1, 0.0, 0.02);
volumeControl.gain.setTargetAtTime(0, 0.1, 0.2);

// Мы собираемся посылать звук в стандартное место назначения:
// динамики пользователя.
let speakers = audioContext.destination;

// Подключить каждую исходную ноту к регулятору громкости.
oscillators.forEach(o => o.connect(volumeControl));

// И подключить выход регулятора громкости к динамикам.
volumeControl.connect(speakers);

// Теперь начать воспроизведение звуков и позволить ему
// длиться 1.25 секунды.
let startTime = audioContext.currentTime;
```

```
let stopTime = startTime + 1.25;
oscillators.forEach(o => {
   o.start(startTime);
   o.stop(stopTime);
});
// Если мы хотим создать последовательность звуков,
// то можем использовать обработчики событий.
oscillators[0].addEventListener("ended", () => {
   // Этот обработчик событий вызывается,
   // когда нота заканчивает воспроизведение.
});
```

15.10. Местоположение, навигация и хронология

Свойство `location` объектов `Window` и `Document` ссылается на объект `Location`, который представляет URL текущего документа, отображаемого в окне, и также предлагает API-интерфейс для загрузки новых документов в окно.

Объект `Location` очень похож на объект `URL` (см. раздел 11.9), и для доступа к различным частям URL текущего документа вы можете использовать свойства вроде `protocol`, `hostname`, `port` и `path`. Свойство `href` возвращает полный URL в виде строки, как делает метод `toString()`.

Интерес вызывают свойства `hash` и `search` объекта `Location`. Свойство `hash` возвращает порцию "идентификатора фрагмента" URL при ее наличии: символ решетки (#), за которым следует идентификатор элемента. Свойство `search` похоже. Оно возвращает порцию URL, которая начинается с вопросительного знака: часто какой-то строки запроса. Как правило, эта порция URL применяется для параметризации URL и предоставляет способ внедрения в него аргументов. Хотя эти аргументы обычно предназначены для сценариев, запускаемых на сервере, нет никаких причин, которые препятствовали бы их использованию внутри страниц с поддержкой JavaScript.

Объекты `URL` имеют свойство `searchParams`, которое является проанализированным представлением свойства `search`. Объект `Location` не имеет свойства `searchParams`, но если вы хотите проанализировать `window.location.search`, то можете просто создать объект `URL` из объекта `Location` и затем применять `searchParams` `URL`:

```
let url = new URL(window.location);
let query = url.searchParams.get("q");
let numResults = parseInt(url.searchParams.get("n") || "10");
```

В дополнение к объекту `Location`, на который вы можете ссылаться через `window.location` или `document.location`, и конструктору `URL()`, который мы использовали ранее, браузеры также определяют свойство `document.URL`. Удивительно, но его значение — не объект `URL`, а всего лишь строка, которая хранит URL текущего документа.

15.10.1. Загрузка новых документов

Если вы присвоите строку свойству window.location или document.location, тогда такая строка интерпретируется как URL и браузер выполнит загрузку по нему, замещая текущий документ новым:

```
window.location = "http://www.oreilly.com";  // Зайдите купить
                                             // несколько книг!
```

Вы также можете присваивать свойству location относительные URL. Они распознаются относительно текущего URL:

```
document.location = "page2.html";   // Загрузить следующую страницу.
```

Пустой идентификатор фрагмента является специальным видом относительного URL, который не заставляет браузер загружать новый документ, а просто инициирует прокрутку, так что элемент документа с атрибутом id или name, соответствующим фрагменту, становится видимым в верхней части окна браузера. В качестве особого случая идентификатор фрагмента #top вынуждает браузер перейти в начало документа (предполагая, что нет элементов с атрибутом id="top"):

```
location = "#top";                  // Перейти в начало документа.
```

Индивидуальные свойства объекта Location допускают запись, а их установка изменяет URL местоположения и также заставляет браузер загрузить новый документ (или в случае свойства hash передвинуться внутри текущего документа):

```
document.location.path = "pages/3.html"; // Загрузить новую страницу.
document.location.hash = "TOC";          // Выполнить прокрутку
                                         // до оглавления.
location.search = "?page=" + (page+1);   // Перезагрузить с новой
                                         // строкой запроса.
```

Загрузить новую страницу можно также за счет передачи новой строки методу assign() объекта Location. Тем не менее, это то же самое, что и присваивание строки свойству location, а потому не особо интересно.

С другой стороны, метод replace() объекта Location довольно-таки полезен. Когда вы передаете строку методу replace(), она интерпретируется как URL и вынуждает браузер загрузить новую страницу, как делает метод assign(). Разница в том, что replace() заменяет текущий документ в хронологии браузера. Если сценарий в документе A устанавливает свойство location или вызывает метод assign() для загрузки документа B и затем пользователь щелкает на кнопке перехода на предыдущую страницу, тогда браузер возвратится обратно к документу A. Если взамен вы применяете replace(), то документ A удаляется из хронологии браузера, и когда пользователь щелкает на кнопке перехода на предыдущую страницу, браузер возвращается к тому документу, который отображался перед документом A.

Когда сценарий загружает новый документ безусловным образом, метод replace() будет лучшим вариантом, чем assign(). Иначе кнопка перехода на

предыдущую страницу вернет браузер на первоначальный документ и тот же самый сценарий снова загрузит новый документ. Предположим, что у вас есть усовершенствованная посредством JavaScript версия вашей страницы и статическая версия, в которой JavaScript не используется. Если вы определили, что браузер пользователя не поддерживает API-интерфейсы веб-платформы, с которыми вы хотите работать, тогда вы могли бы применить `location.replace()` для загрузки статической версии:

```
// Если браузер не поддерживает необходимые нам API-интерфейсы
// для JavaScript, тогда перенаправить его на статическую страницу,
// не использующую JavaScript.
if (!isBrowserSupported()) location.replace("staticpage.html");
```

Обратите внимание, что URL, переданный методу `replace()`, является относительным. Относительные URLs интерпретируются относительно страницы, в которой они появляются, точно так же, как если бы они использовались в гиперссылке.

Помимо методов `assign()` и `replace()` в объекте `Location` также определен метод `reload()`, который просто заставляет браузер перезагрузить документ.

15.10.2. Хронология просмотра

Свойство `history` объекта `Window` ссылается на объект `History` для окна, который моделирует хронологию просмотра окна в виде списка документов и состояний документов. Свойство `length` объекта `History` указывает количество элементов в списке хронологии просмотра, но по причинам, связанным с безопасностью, сценариям не разрешен доступ к сохраненным URL. (Если бы доступ был разрешен, тогда любые сценарии могли бы подсматривать вашу хронологию просмотра.)

Объект `History` имеет методы `back()` и `forward()`, поведение которых похоже на действие кнопок перехода на предыдущую и на следующую страницу браузера: они вынуждают браузер переходить на один шаг назад и вперед в хронологии просмотра. Третий метод, `go()`, принимает целочисленный аргумент и может пропускать любое количество страниц вперед (для положительного значения аргумента) или назад (для отрицательного значения аргумента) в списке хронологии:

```
history.go(-2);    // Перейти на два шага назад, подобно двукратному
                   // щелчку на кнопке перехода на предыдущую страницу.
history.go(0);     // Другой способ перезагрузки текущей страницы.
```

Если окно содержит дочерние окна (такие как элементы `<iframe>`), тогда хронологии просмотра дочерних окон по времени перемежаются с хронологией главного окна. Это означает, что вызов `history.back()` (например) для главного окна может заставить одно из дочерних окон перейти назад к предыдущему отображаемому документу, но главное окно останется в своем текущем состоянии.

Обсуждаемый здесь объект History относится к раннему периоду существования веб-сети, когда документы были пассивными и все вычисления выполнялись на сервере. В наши дни веб-приложения часто генерируют или загружают содержимое динамически и отображают новые состояния приложения без фактической загрузки новых документов. Приложения такого рода должны самостоятельно управлять хронологией, если желательно, чтобы пользователь имел возможность применять кнопки перехода на предыдущую и на следующую страницу (или эквивалентные жесты) для интуитивно понятной навигации по состояниям приложения. Решить задачу можно двумя способами, которые описаны в последующих двух подразделах.

15.10.3. Управление хронологией с помощью событий "hashchange"

Первая методика управления хронологией задействует свойство location.hash и событие "hashchange". Ниже перечислены ключевые факты, которые необходимо знать, чтобы понять данную методику.

- Свойство location.hash устанавливает идентификатор фрагмента URL и традиционно используется для указания идентификатора раздела документа, до которого нужно выполнить прокрутку. Но свойство location.hash не обязано быть идентификатором элемента: вы можете установить его в любую строку. При условии, что не существует элемента с идентификатором, совпадающим с указанной в свойстве hash строкой, браузер прокрутку выполнять не будет.
- Установка свойства location.hash приводит к обновлению URL, отображаемому в поле местоположения и, что крайне важно, к добавлению записи в хронологию браузера.
- Всякий раз, когда идентификатор фрагмента документа изменяется, браузер инициирует событие "hashchange" в объекте Window. Если вы явно устанавливаете свойство location.hash, тогда сгенерируется событие "hashchange". И, как уже упоминалось, такое изменение объекта Location приводит к созданию новой записи в хронологии просмотра браузера. Следовательно, если пользователь щелкнет на кнопке перехода на предыдущую страницу, то браузер возвратится к своему предыдущему URL, который был перед установкой location.hash. Но это значит, что идентификатор фрагмента снова изменился, а потому инициируется еще одно событие "hashchange". Таким образом, при условии, что вы сумеете создать уникальные идентификаторы фрагментов для каждого возможного состояния приложения, события "hashchange" будут уведомлять вас в случае перехода пользователя назад и вперед по хронологии просмотра.

Для применения такого механизма управления хронологией вы должны иметь возможность кодировать информацию состояния, необходимую при визуализации "страницы" вашего приложения, в виде относительно короткой строки текста, которую можно было бы использовать в качестве идентификато-

ра фрагмента. И вам потребуется написать функцию для преобразования состояния страницы в строку и еще одну функцию для разбора строки и воссоздания состояния страницы, которое она представляет.

После того, как вы напишете указанные функции, остальное делается легко. Определите функцию `window.onhashchange` (или зарегистрируйте прослушиватель событий `"hashchange"` с помощью `addEventListener()`), которая читает `location.hash`, преобразует прочитанную строку в представление вашего приложения и затем предпринимает любые действия, необходимые для отображения нового состояния приложения.

Когда пользователь взаимодействует с вашим приложением (скажем, щелкает на ссылке), так что вынуждает приложение войти в новое состояние, не визуализируйте новое состояние напрямую. Взамен закодируйте желаемое новое состояние как строку и установите свойство `location.hash` в эту строку. В результате инициируется событие `"hashchange"` и ваш обработчик для данного события отобразит новое состояние. Применение такой обходной методики гарантирует, что новое состояние вставляется в хронологию просмотра, а потому кнопки перехода на предыдущую и на следующую страницу продолжают работать.

15.10.4. Управление хронологией с помощью метода `pushState()`

Вторая методика для управления хронологией несколько сложнее, но менее похожа на хакерский прием, чем использование события `"hashchange"`. Эта более надежная методика для управления хронологией основана на методе `history.pushState()` и событии `"popstate"`. Когда веб-приложение входит в новое состояние, оно вызывает `history.pushState()` для добавления объекта, представляющего состояние, в хронологию браузера. Если пользователь затем щелкнет на кнопке перехода на предыдущую страницу, то браузер сгенерирует событие `"popstate"` с копией объекта сохраненного состояния, а приложение задействует этот объект для воссоздания своего предыдущего состояния. Помимо объекта сохраненного состояния приложения также могут сохранять URL с каждым состоянием, что важно, если вы хотите предоставить пользователям возможность создавать закладки и делиться ссылками на внутренние состояния приложений.

Первым аргументом `pushState()` является объект, который содержит всю информацию состояния, необходимую для восстановления текущего состояния документа. Такой объект сохраняется с применением алгоритма *структурированного клонирования* HTML, который более универсален, чем `JSON.stringify()`, и способен поддерживать объекты `Map`, `Set` и `Date`, а также типизированные массивы и объекты `ArrayBuffer`.

Второй аргумент планировался быть строкой заголовка для состояния, но большинство браузеров его не поддерживают, и вы должны просто передавать в нем пустую строку. Третий аргумент — необязательный URL, который будет отображаться в поле местоположения немедленно и также в случае, если пользователь возвращается в это состояние посредством кнопок перехода на

предыдущую и на следующую страницу. Относительные URL распознаются относительно текущего местоположения документа. Связывание URL с каждым состоянием позволяет пользователю создавать закладки для внутренних состояний вашего приложения. Однако помните, что если пользователь сохраняет закладку и затем посещает ее днем позже, то вы не получите событие "popstate" о таком посещении: вам придется восстанавливать состояние приложения, анализируя URL.

Алгоритм структурированного клонирования

Метод `history.pushState()` не использует функцию `JSON.stringify()` (см. раздел 11.6) для сериализации данных состояния. Взамен он (и другие API-интерфейсы браузера, обсуждаемые позже) применяет более надежную методику сериализации, которая известна как алгоритм структурированного клонирования, определенный стандартом HTML.

Алгоритм структурированного клонирования способен сериализировать все, что может сериализировать `JSON.stringify()`, но вдобавок он делает возможной сериализацию большинства других типов JavaScript, включая `Map`, `Set`, `Date`, `RegExp` и типизированные массивы, а также умеет обрабатывать структуры данных, которые содержат циклические ссылки. Тем не менее, алгоритм структурированного клонирования не может сериализировать функции или классы. При клонировании объектов он не копирует объект-прототип, методы получения и установки или неперечислимые свойства. Наряду с тем, что алгоритм структурированного клонирования способен клонировать большинство встроенных типов JavaScript, он не может копировать типы, определяемые средой размещения, такие как объекты `Element` документа.

Таким образом, объект состояния, передаваемый методу `history.pushState()`, не обязан ограничиваться объектами, массивами и элементарными значениями, которые поддерживает функция `JSON.stringify()`. Однако имейте в виду, что если вы передадите экземпляр класса, который определили сами, то этот экземпляр будет сериализироваться как обыкновенный объект JavaScript и утратит свой прототип.

Помимо метода `pushState()` в объекте `History` также определен метод `replaceState()`, который принимает те же самые аргументы, но замещает текущее состояние хронологии, а не добавляет новое состояние в хронологию просмотра. При первой загрузке приложения, в котором используется метод `pushState()`, часто бывает полезно вызвать метод `replaceState()`, чтобы определить объект состояния для начального состояния приложения.

Когда пользователь переходит к сохраненному состоянию хронологии с применением кнопок перехода на предыдущую и на следующую страницу, браузер инициирует событие "popstate" в объекте `Window`. Ассоциированный с событием объект события имеет свойство по имени `state`, которое содержит копию (еще один структурированный клон) объекта состояния, переданного методу `pushState()`.

В примере 15.9 приведено простое веб-приложение игры в угадывание чисел, представленное на рис. 15.15, в котором используется метод `pushState()` для сохранения своей хронологии, давая пользователю возможность "вернуться", чтобы пересмотреть или повторить свои предположения.

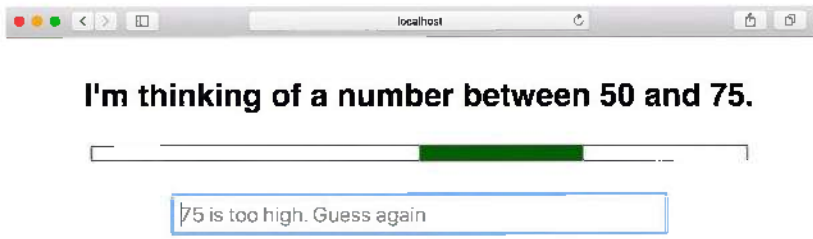


Рис. 15.15. Игра в угадывание чисел

Пример 15.9. Управление хронологией с помощью `pushState()`

```
<html><head><title>I'm thinking of a number...</title>
<style>
body { height: 250px; display: flex; flex-direction: column;
     align-items: center; justify-content: space-evenly; }
#heading { font: bold 36px sans-serif; margin: 0; }
#container { border: solid black 1px; height: 1em; width: 80%; }
#range { background-color: green; margin-left: 0%; height: 1em; width: 100%; }
#input { display: block; font-size: 24px; width: 60%; padding: 5px; }
#playagain { font-size: 24px; padding: 10px; border-radius: 5px; }
</style>
</head>
<body>
<h1 id="heading">I'm thinking of a number...</h1>
<!-- Визуальное представление чисел, которые не были исключены. -->
<div id="container"><div id="range"></div></div>
<!-- Место, где пользователь вводит свое предположение. -->
<input id="input" type="text">
<!-- Кнопка, которая перезагружает страницу без строки поиска.
     Скрыта вплоть до конца игры. -->
<button id="playagain" hidden onclick="location.search='';">Play Again
</button>
<script>
/**
 * Экземпляр этого класса GameState представляет внутреннее состояние нашей
 * игры в угадывание чисел. В классе определены статические фабричные методы
 * для инициализации состояния игры из различных источников, метод для
 * обновления состояния на основе нового предположения и метод для изменения
 * документа на основе текущего состояния.
 */
class GameState {
   // Это фабричная функция для создания новой игры.
   static newGame() {
      let s = new GameState();
      s.secret = s.randomInt(0, 100);  // Целое число: 0 < n < 100.
```

```
    s.low = 0;                  // Предположения должны быть больше, чем это.
    s.high = 100;               // Предположения должны быть меньше, чем это.
    s.numGuesses = 0;           // Сколько предположений было сделано.
    s.guess = null;             // Каким было последнее предположение.
    return s;
}

// Когда мы сохранили состояние игры с помощью history.pushState(),
// оно будет сохраненным простым объектом JavaScript, не экземпляром
// GameState. Таким образом, эта фабричная функция воссоздает объект
// GameState на основе простого объекта, который мы получаем из события
// "popstate".
static fromStateObject(stateObject) {
    let s = new GameState();
    for(let key of Object.keys(stateObject)) {
        s[key] = stateObject[key];
    }
    return s;
}

// Чтобы позволить создавать закладки, нам необходимо иметь возможность
// кодировать состояние любой игры в виде URL. Это легко сделать
// посредством URLSearchParams.
toURL() {
    let url = new URL(window.location);
    url.searchParams.set("l", this.low);
    url.searchParams.set("h", this.high);
    url.searchParams.set("n", this.numGuesses);
    url.searchParams.set("g", this.guess);
    // Обратите внимание, что мы не можем закодировать секретное число
    // в URL, иначе секрет будет выдан. Если пользователь создал закладку
    // для страницы с этими параметрами и затем возвращается на нее, тогда
    // мы просто выбираем новое случайное число между low и high.
    return url.href;
}

// Это фабричная функция, которая создает новый объект GameState
// и инициализирует его из указанного URL. Если URL не содержит
// ожидаемые параметры или они неправильно оформлены,
// то она просто возвращает null.
static fromURL(url) {
    let s = new GameState();
    let params = new URL(url).searchParams;
    s.low = parseInt(params.get("l"));
    s.high = parseInt(params.get("h"));
    s.numGuesses = parseInt(params.get("n"));
    s.guess = parseInt(params.get("g"));

    // Если в URL пропущен любой необходимый параметр или параметры
    // не разбираются в целые числа, тогда возвратить null.
    if (isNaN(s.low) || isNaN(s.high) ||
        isNaN(s.numGuesses) || isNaN(s.guess)) {
        return null;
    }
```

```
    // Каждый раз, когда мы восстанавливаем игру из URL, выбирать
    // новое секретное число из правильного диапазона.
    s.secret = s.randomInt(s.low, s.high);
    return s;
}

// Возвратить целое число, min < n < max.
randomInt(min, max) {
    return min + Math.ceil(Math.random() * (max - min - 1));
}

// Изменить документ для отображения текущего состояния игры.
render() {
    let heading = document.querySelector("#heading"); // Элемент <h1> вверху.
    let range = document.querySelector("#range");   // Отобразить диапазон
                                                    // для предположений.
    let input = document.querySelector("#input");   // Поле для ввода
                                                    // предположений.
    let playagain = document.querySelector("#playagain");

    // Обновить заголовок и заглавие документа.
    heading.textContent = document.title =
        `I'm thinking of a number between ${this.low} and ${this.high}.`;

    // Обновить визуальный диапазон чисел.
    range.style.marginLeft = `${this.low}%`;
    range.style.width = `${(this.high-this.low)}%`;

    // Обеспечить, чтобы поле ввода было пустым и в фокусе.
    input.value = "";
    input.focus();

    // Отобразить ответ на основе последнего предположения пользователя.
    // Заполнитель при вводе будет показан, т.к. мы сделали поле ввода
    // пустым.
    if (this.guess === null) {
        input.placeholder = "Type your guess and hit Enter";
                        // Введите свое предположение и нажмите Enter
    } else if (this.guess < this.secret) {
        input.placeholder = `${this.guess} is too low. Guess again`;
                        // Предположение слишком малое. Повторите попытку
    } else if (this.guess > this.secret) {
        input.placeholder = `${this.guess} is too high. Guess again`;
                        // Предположение слишком большое. Повторите попытку
    } else {
        input.placeholder = document.title = `${this.guess} is correct!`;
                        // Предположение корректно!
        heading.textContent = `You win in ${this.numGuesses} guesses!`;
                        // Вы выиграли за столько-то попыток!
        playagain.hidden = false;
    }
}

// Обновить состояние игры на основе того, что пользователь предположил.
// Возвратить true, если состояние было обновлено, и false в противном случае
updateForGuess(guess) {
```

```
        // Если предположение - число и находится в правильном диапазоне.
        if ((guess > this.low) && (guess < this.high)) {
            // Обновить объект состояния на основе этого предположения.
            if (guess < this.secret) this.low = guess;
            else if (guess > this.secret) this.high = guess;
            this.guess = guess;
            this.numGuesses++;
            return true;
        }
        else {  // Неправильное предположение: уведомить пользователя,
                // но не обновлять состояние.
            alert(`Please enter a number greater than ${
                  this.low} and less than ${this.high}`);
               // Введите число в нужном диапазоне
            return false;
        }
    }
}
// Имея определенный класс GameState, запуск игры в работу сводится просто
// к инициализации, обновлению, сохранению и визуализации объекта состояния
// в подходящие моменты времени.

// При первой загрузке мы пытаемся получить состояние игры из URL, и если это
// не удается, то взамен мы начинаем новую игру. Таким образом, если пользователь
// сделал закладку для игры, то игра может быть восстановлена из URL. Но если мы
// загружаем страницу без параметров запроса, тогда просто получаем новую игру.
let gamestate = GameState.fromURL(window.location) || GameState.newGame();

// Сохранить это начальное состояние игры в хронологии браузера,
// но использовать для начальной страницы replaceState(), а не pushState().
history.replaceState(gamestate, "", gamestate.toURL());

// Отобразить это начальное состояние.
gamestate.render();

// Когда пользователь вводит предположение, обновить состояние игры
// на основе его предположения, затем сохранить новое состояние
// в хронологии браузера и визуализировать новое состояние.
document.querySelector("#input").onchange = (event) => {
    if (gamestate.updateForGuess(parseInt(event.target.value))) {
        history.pushState(gamestate, "", gamestate.toURL());
    }
    gamestate.render();
};

// Если пользователь перемещается назад или вперед по хронологии,
// то мы будем получать в объекте window событие "popstate" с копией
// объекта состояния, сохраненного с помощью pushState().
// Когда это происходит, визуализировать новое состояние.
window.onpopstate = (event) => {
    gamestate = GameState.fromStateObject(event.state); // Восстановить
                                                        // состояние
    gamestate.render();                                 // и отобразить его.
};
</script>
</body></html>
```

15.11. Взаимодействие с сетью

Каждый раз, когда вы загружаете веб-страницу, браузер делает сетевые запросы с применением протоколов HTTP и HTTPS для HTML-файла, а также для изображений, шрифтов, сценариев и таблиц стилей, от которых зависит HTML-файл. Но в дополнение к возможности делать сетевые запросы в ответ на пользовательские действия веб-браузеры также предоставляют API-интерфейсы JavaScript для работы с сетью.

В настоящем разделе раскрываются три API-интерфейсы для взаимодействия с сетью.

- Метод `fetch()` определяет API-интерфейс, основанный на `Promise`, для создания HTTP- и HTTPS-запросов. API-интерфейс `fetch()` делает базовые запросы GET простыми, но обладает всесторонним набором средств, который также поддерживает практически любой возможный сценарий использования HTTP.
- API-интерфейс SSE (Server-Sent Events — события, посылаемые сервером) представляет собой удобный, основанный на событиях интерфейс для методик "длинного опроса" HTTP, при которых веб-сервер удерживает сетевое подключение открытым, так что он может посылать данные клиенту всякий раз, когда пожелает.
- Веб-сокеты — это сетевой протокол, который не является HTTP, но предназначен для взаимодействия с HTTP. Он определяет асинхронный API-интерфейс передачи сообщений, где клиенты и серверы могут посылать и принимать сообщения друг от друга способом, похожим на сетевые сокеты TCP.

15.11.1. `fetch()`

Для базовых HTTP-запросов применение `fetch()` сводится к процессу, состоящему из трех шагов.

1. Вызвать `fetch()`, передавая URL, содержимое которого нужно извлечь.
2. Получить объект ответа, который асинхронно возвратился на шаге 1, когда HTTP-ответ начинает поступать, и вызвать метод объекта ответа, чтобы затребовать тело ответа.
3. Получить объект тела, который асинхронно возвратился на шаге 2, и обработать его любым желаемым способом.

API-интерфейс `fetch()` полностью основан на `Promise`, и здесь есть два асинхронных шага, а потому при использовании `fetch()` обычно ожидаются два вызова `then()` или два выражения `await`. (Если вы забыли, что это такое, тогда перечитайте главу 13, прежде чем продолжать изучение текущего подраздела.)

Вот как выглядит запрос `fetch()`, если вы применяете `then()` и ожидаете, что ответ сервера на ваш запрос будет иметь формат JSON:

```
fetch("/api/users/current")     // Сделать HTTP- или HTTPS-запрос GET.
  .then(response => response.json()) // Разобрать его тело в объект JSON
  .then(currentUser => {    // Затем обработать этот разобранный объект.
    displayUserInfo(currentUser);
  });
```

Ниже приведен похожий запрос, сделанный с использованием ключевых слов `async` и `await` к API-интерфейсу, который возвращает вместо объекта JSON простую строку:

```
async function isServiceReady() {
  let response = await fetch("/api/service/status");
  let body = await response.text();
  return body === "ready";
}
```

Если вы понимаете эти два примера кода, то знаете 80% того, что нужно знать для применения API-интерфейса `fetch()`. В последующих подразделах будет показано, как делать запросы и принимать ответы, которые несколько сложнее продемонстрированных выше.

Прощай, `XMLHttpRequest`!

API-интерфейс `fetch()` замещает собой причудливый и обманчиво названный API-интерфейс `XMLHttpRequest` (который не имеет ничего общего с XML). Вы все еще можете сталкиваться с `XMLHttpRequest` (или его аббревиатурой XHR) в существующем коде, но нет никаких причин использовать его в новом коде и потому в настоящей главе он не документируется. Тем не менее, один пример применения `XMLHttpRequest` в этой книге есть, и вы можете обратиться в подраздел 13.1.3, если хотите увидеть пример взаимодействия с сетью посредством JavaScript в старом стиле.

Коды состояний HTTP, заголовки ответов и ошибки сети

Трехшаговый процесс `fetch()`, описанный в подразделе 15.11.1, исключает весь код обработки ошибок. Вот более реалистичная версия:

```
fetch("/api/users/current") // Сделать HTTP- или HTTPS-запрос GET.
  .then(response => { //Когда мы получим ответ, сначала проверить его
    if (response.ok &&   // на наличие кода успеха и ожидаемого типа.
      response.headers.get("Content-Type") === "application/json") {
      return response.json(); //Возвратить объект Promise для тела
    } else {
      throw new Error(         // Или сгенерировать ошибку.
        `Unexpected response status ${response.status} or
content type`
      );
    }
  })
  .then(currentUser => {           // Когда объект Promise
                                   // из response.json() разрешается,
    displayUserInfo(currentUser); //делать что-то с разобранным телом
  })
```

```
.catch(error => {    // Или если что-то пошло не так,
                     // тогда вывести сообщение об ошибке.
  // Если браузер пользователя работает автономно,
  // тогда сам вызов fetch() будет отклонен.
  // Если сервер возвращает неверный ответ,
  // тогда мы генерируем ошибку выше.
  console.log("Error while fetching current user:", error);
});
```

Объект `Promise`, возвращенный из `fetch()`, разрешается в объект `Response`. Свойство `status` объекта `Response` хранит код состояния HTTP, такой как 200 для успешных запросов или 404 для запросов "Not Found" (не найдено). (Свойство `statusText` дает стандартный текст на английском, сопровождающий числовой код состояния.) Удобно то, что свойство `ok` объекта `Response` равно `true`, если `status` содержит 200 или код между 200 и 299, и `false` для любого другого кода.

Функция `fetch()` разрешает свой объект `Promise`, когда ответ сервера начинает поступать, как только состояние и заголовки ответа HTTP становятся доступными, но обычно до поступления полного тела ответа. Несмотря на то что тело пока недоступно, на втором шаге процесса извлечения вы можете исследовать заголовки. Свойство `headers` объекта `Response` — это объект `Headers`. Используйте его метод `has()` для проверки наличия заголовка или его метод `get()`, чтобы получить значение заголовка. Имена заголовков HTTP нечувствительны к регистру, так что можете передавать указанным методам имена заголовков в нижнем или смешанном регистре.

Объект `Headers` также является итерируемым, если вас когда-либо заинтересует данный факт:

```
fetch(url).then(response => {
  for(let [name,value] of response.headers) {
    console.log(`${name}: ${value}`);
  }
});
```

Если веб-сервер отвечает на ваш запрос `fetch()`, тогда возвращенный объект `Promise` будет удовлетворен с объектом `Response`, даже когда ответом сервера была ошибка 404 Not Found (не найдено) или 500 Internal Server Error (внутренняя ошибка сервера). Функция `fetch()` отклоняет возвращенный объект `Promise`, только если он вообще не может связаться с веб-сервером. Такое случается, когда компьютер пользователя не подключен к сети, сервер не реагирует на запросы или в URL указано несуществующее имя хоста. Поскольку подобные вещи могут происходить с любым сетевым запросом, рекомендуется включать в любой вызов `fetch()` конструкцию `.catch()`.

Установка параметров запроса

Временами, делая запрос, вы хотите передавать дополнительные параметры вместе с URL. Для этого в конец URL после `?` понадобится добавить пары имя/значение. Классы `URL` и `URLSearchParams` (которые обсуждались в разделе 11.9)

упрощают конструирование URL в такой форме. Функция fetch() принимает в своем первом аргументе объекты URL, а потому вот как можно включить в запрос fetch() параметры запроса:

```
async function search(term) {
    let url = new URL("/api/search");
    url.searchParams.set("q", term);
    let response = await fetch(url);
    if (!response.ok) throw new Error(response.statusText);
    let resultsArray = await response.json();
    return resultsArray;
}
```

Установка заголовков запроса

Иногда в запросах fetch() необходимо устанавливать заголовки. Скажем, если вы делаете запросы к API-интерфейсу для веб-сети, который требует учетных данных, тогда вам может потребоваться включить заголовок Authorization, содержащий эти учетные данные. В таком случае вы можете применять версию fetch() с двумя аргументами. Как и ранее, в первом аргументе передается строка или объект URL, который задает URL, подлежащий извлечению. Во втором аргументе указывается объект, который предоставляет дополнительные параметры, включая заголовки запроса:

```
let authHeaders = new Headers();
// Не используйте аутентификацию Basic, если это не подключение HTTPS.
authHeaders.set("Authorization",
                `Basic ${btoa(`${username}:${password}`)}`);
fetch("/api/users/", { headers: authHeaders })
    .then(response => response.json())  // Обработка ошибок опущена...
    .then(usersList => displayAllUsers(usersList));
```

Существует несколько других параметров, которые могут быть указаны во втором аргументе функции fetch(), и вы увидите их позже. Вместо передачи двух аргументов функции fetch() можно передать те же два аргумента конструктору Request() и затем передать результирующий объект Request вызову fetch():

```
let request = new Request(url, { headers });
fetch(request).then(response => ...);
```

Разбор тела ответа

В продемонстрированном ранее трехшаговом процессе fetch() второй шаг заканчивается вызовом метода json() или text() объекта Response и возвращением объекта Promise, который возвращают упомянутые методы. Третий шаг начинается, когда объект Promise разрешается с телом ответа, разобранным как объект JSON или просто как строка текста.

Вероятно, описанные два сценария являются самыми распространенными, но не единственными способами получения тела ответа веб-сервера.

Помимо json() и text() в объекте Response есть также описанные далее методы.

- **`arrayBuffer()`**. Этот метод возвращает объект `Promise`, который разрешается в объект `ArrayBuffer`. Он удобен, когда ответ содержит двоичные данные. Вы можете использовать `ArrayBuffer` для создания типизированного массива (см. раздел 11.2) или объекта `DataView` (см. подраздел 11.2.5), из которого можно читать двоичные данные.
- **`blob()`**. Этот метод возвращает объект `Promise`, который разрешается в объект `Blob`. Объекты `Blob` в книге подробно не рассматриваются, но само имя означает "Binary Large Object" (большой двоичный объект), и они полезны, когда вы ожидаете получить крупные объемы двоичных данных. Если вы затребуете тело ответа как объект `Blob`, тогда реализация браузера может передать данные ответа во временный файл и затем возвратить объект `Blob`, который представляет этот временный файл. Следовательно, объекты `Blob` не поддерживают произвольный доступ к телу ответа, как в `ArrayBuffer`. Имея в своем распоряжении объект `Blob`, вы можете создать объект `URL`, который ссылается на него посредством `URL.createObjectURL()`, или применить основанный на событиях API-интерфейс `FileReader` для асинхронного получения содержимого объекта `Blob` в виде строки или объекта `ArrayBuffer`. На момент написания главы некоторые браузеры также определяли основанные на `Promise` методы `text()` и `arrayBuffer()`, предлагающие более прямой путь для получения содержимого объекта `Blob`.
- **`formData()`**. Этот метод возвращает объект `Promise`, который разрешается в объект `FormData`. Вы должны использовать метод `formData()`, если ожидаете, что тело `Response` должно быть закодировано в формате `"multipart/form-data"`. Такой формат распространен в запросах `POST`, посылаемых серверу, но редко встречается в ответах сервера и потому применяется нечасто.

Потоковая передача тела ответа

В дополнение к пяти методам ответа, которые асинхронно возвращают полное тело ответа в определенной форме, имеется также возможность потоковой передачи тела ответа, что удобно, когда нужно организовать какую-то обработку, допускающую выполнение над частями тела ответа по мере их поступления по сети. Но потоковая передача также полезна, если вы хотите отображать индикатор хода работ, чтобы пользователь мог видеть продвижение загрузки.

Свойство `body` объекта `Response` — это объект `ReadableStream`. Если вы уже вызвали метод ответа вроде `text()` или `json()`, который читает, проводит разбор и возвращает тело, тогда `bodyUsed` будет иметь значение `true`, указывая на то, что поток `body` был прочитан. Однако если свойство `bodyUsed` равно `false`, то поток пока еще не прочитан. В таком случае вы можете вызвать `getReader()` на `response.body`, чтобы получить объект средства чтения потока и затем использовать метод `read()` этого объекта для асинхронного чтения порций текста из потока. Метод `read()` возвращает объект `Promise`, который разрешается в объект со свойствами `done` и `value`. Свойство `done` будет равно

true, если тело было прочитано целиком или если поток был закрыт. А свойство value будет либо следующей порцией в виде объекта Uint8Array, либо undefined, если порций больше нет.

API-интерфейс потоковой передачи относительно прямолинеен, если вы применяете ключевые слова async и await, но удивительно сложен, когда вы попытаетесь использовать его с низкоуровневыми объектами Promise. В примере 15.10 демонстрируется работа с данным API-интерфейсом при определении функции streamBody(). Предположим, что вы хотите загрузить крупный файл JSON и сообщать пользователю о продвижении. Сделать это с помощью метода json() объекта Response не удастся, но вы могли бы применить функцию streamBody(), как показано ниже (допускается, что функция updateProgress() определена для установки атрибута value в HTML-элементе <progress>):

```
fetch('big.json')
    .then(response => streamBody(response, updateProgress))
    .then(bodyText => JSON.parse(bodyText))
    .then(handleBigJSONObject);
```

Реализация функции streamBody() приведена в примере 15.10.

Пример 15.10. Потоковая передача тела ответа из запроса fetch()

```
/**
 * Асинхронная функция для потоковой передачи тела объекта Response,
 * полученного из запроса fetch(). Принимает в первом аргументе
 * объект Response и за ним два необязательных обратных вызова.
 *
 * Если вы указали функцию в качестве второго аргумента, то этот обратный
 * вызов reportProgress будет вызываться один раз для каждой получаемой порции.
 * В первом аргументе передается общее количество байтов, полученных до сих пор.
 * Во втором аргументе передается число между 0 и 1, которое указывает,
 * насколько загрузка завершена. Однако если объект Response не имеет
 * заголовка "Content-Length", тогда вторым аргументом всегда будет NaN.
 *
 * Если вы хотите обрабатывать данные в порциях, когда они прибывают,
 * то укажите функцию в третьем аргументе. Порции будут передаваться
 * этому обратному вызову
 * processChunk как объекты Uint8Array.
 *
 * streamBody() возвращает объект Promise, который разрешается в строку.
 * Если обратный вызов processChunk был предоставлен, тогда эта строка
 * является сцеплением значений, возвращенных processChunk. Иначе строка
 * будет сцеплением значений порций, преобразованных в строки UTF-8.
 */
async function streamBody(response, reportProgress, processChunk) {
    // Ожидаемое количество байтов или NaN, если нет заголовка.
    let expectedBytes = parseInt(response.headers.get("Content-Length"));
    let bytesRead = 0;                  // Сколько байтов получено до сих пор.
```

```
    let reader = response.body.getReader();    // Читать байты с помощью этой
                                               // функции.
    let decoder = new TextDecoder("utf-8"); //Для преобразования байтов в текст
    let body = "";                          // Текст, прочитанный до сих пор.

    while(true) {                           // Цикл, пока не будет выход ниже.
      let {done, value} = await reader.read();  // Читать порцию.

      if (value) {                          // Если мы получили байтовый массив:
        if (processChunk) {                 // обработать байты, когда обратный
          let processed = processChunk(value);  // вызов был передан.
          if (processed) {
            body += processed;
          }
        } else {                            //В противном случае преобразовать байты
          body += decoder.decode(value, {stream: true});  // в текст.
        }

        if (reportProgress) {               // Если обратный вызов продвижения
          bytesRead += value.length;  // был передан, тогда вызвать его.
          reportProgress(bytesRead, bytesRead / expectedBytes);
        }
      }
      if (done) {                           // Если это последняя порция,
        break;                              // тогда выйти из него.
      }
    }

    return body;    // Возвратить накопленный текст тела.
}
```

На момент написания главы API-интерфейс потоковой передачи был новым и, как ожидается, он будет развиваться. В частности, есть планы сделать объекты `ReadableStream` асинхронно итерируемыми, чтобы их можно было использовать с циклами `for/await` (см. подраздел 13.4.1).

Указание метода запроса и тела запроса

В каждом показанном до сих пор примере `fetch()` мы делали HTTP- или HTTPS-запрос GET. Если вы хотите применить другой метод запроса (такой как POST, PUT или DELETE), тогда просто используйте версию `fetch()` с двумя аргументами, передавая ей объект `Options` с параметром `method`:

```
fetch(url,
      { method: "POST" }).then(r => r.json()).then(handleResponse);
```

Запросы POST и PUT обычно имеют тело запроса, содержащее данные, которые должны посылаться серверу. При условии, что свойство `method` не установлено в `"GET"` или `"HEAD"` (методы запроса, не поддерживающие тело), вы можете указывать тело запроса, устанавливая свойство `body` объекта `Options`:

```
fetch(url, {
  method: "POST",
  body: "hello world"
})
```

Когда вы указываете тело запроса, браузер автоматически добавляет к запросу подходящий заголовок "Content-Length". Если тело является строкой, как в предыдущем примере, тогда браузер устанавливает заголовок "Content-Type" в принятое по умолчанию значение "text/plain;charset=UTF-8". Вам может потребоваться переопределить такое значение по умолчанию, если вы указываете строковое тело более специфического типа наподобие "text/html" или "application/json":

```
fetch(url, {
   method: "POST",
   headers: new Headers({"Content-Type": "application/json"}),
   body: JSON.stringify(requestBody)
})
```

Свойство body объекта Options функции fetch() не обязано быть строкой. Если у вас есть двоичные данные в типизированном массиве, в объекте DataView или в объекте ArrayBuffer, то вы можете установить свойство body в такое значение и указать соответствующий заголовок "Content-Type". Если вы располагаете двоичными данными в форме объекта Blob, тогда можете установить body в этот Blob. Объекты Blob имеют свойство type, в котором указывается тип их содержимого, и значение свойства type применяется в качестве значения по умолчанию для заголовка "Content-Type".

С запросами POST довольно часто передается набор параметров имя/значение в теле запроса (вместо их кодирования в порции запроса URL). Есть два способа добиться этого.

- Вы можете указать имена и значения параметров с помощью URLSearchParams (который упоминался ранее в разделе и документировался в разделе 11.9) и затем передать объект URLSearchParams как значение свойства body. В таком случае тело будет установлено в строку, которая выглядит похожей на порцию запроса URL, а заголовок "Content-Type" автоматически будет установлен в "application/x-www-form-urlencoded;charset=UTF-8".
- Если взамен вы укажете имена и значения параметров посредством объекта FormData, тогда тело будет использовать более подробное многоблочное кодирование и заголовок "Content-Type" установится в "multipart/form-data; boundary=..." с уникальной разделительной строкой, что соответствует телу. Применение объекта FormData особенно полезно, когда загружаемые значения являются длинными либо представляют собой объекты File или Blob, каждый из которых может иметь собственный заголовок "Content-Type". Объекты FormData можно создавать и инициализировать значениями, передавая элемент <form> конструктору FormData(). Но вы также можете создавать тела запросов "multipart/form-data" путем вызова конструктора FormData() без аргументов и инициализации пар имя/значения, которые они представляют, с помощью методов set() и append().

Выгрузка файлов с помощью fetch()

Выгрузка файлов из компьютера пользователя на веб-сервер — типичная работа, которую можно выполнить с использованием объекта FormData в качестве тела запроса. Обычный способ получения объекта File предусматривает отображение на веб-странице элемента <input type="file"> и прослушивание событий "change" в этом элементе. Когда возникает событие "change", массив files элемента ввода должен содержать, по меньшей мере, один объект File. Объекты File также доступны через API-интерфейс перетаскивания HTML, который в книге не рассматривается, но вы можете получать файлы из массива dataTransfer.files объекта события, передаваемого прослушивателю событий "drop".

Помните также, что объект File является разновидностью объекта Blob, и временами он может быть удобен для выгрузки объектов Blob. Предположим, что вы написали веб-приложение, которое позволяет пользователю создавать рисунки в элементе <canvas>. Вы можете выгружать рисунки пользователя в виде файлов PNG с помощью следующего кода:

```
// Функция canvas.toBlob() основана на обратном вызове.
// Это оболочка на базе Promise для нее.
async function getCanvasBlob(canvas) {
  return new Promise((resolve, reject) => {
    canvas.toBlob(resolve);
  });
}

// Вот как мы выгружаем файл PNG из холста.
async function uploadCanvasImage(canvas) {
  let pngblob = await getCanvasBlob(canvas);
  let formdata = new FormData();
  formdata.set("canvasimage", pngblob);
  let response = await fetch("/upload", { method: "POST", body: formdata });
  let body = await response.json();
}
```

Запросы между разными источниками

Чаще всего функция fetch() применяется веб-приложениями для запрашивания данных у собственного веб-сервера. Запросы подобного рода известны как запросы из того же самого источника, т.к. URL, передаваемый fetch(), имеет тот же источник (протокол плюс имя хоста плюс порт), что и документ, который содержит сценарий, выполняющий запрос.

По соображениям безопасности веб-браузеры обычно запрещают запросы между разными источниками (хотя существуют исключения для изображений и сценариев). Тем не менее, технология CORS (cross-origin resource sharing — совместное использование ресурсов между разными источниками) делает возможными безопасные запросы между разными источниками. Когда функция fetch() применяется для URL с разными источниками, браузер добавляет к запросу заголовок "Origin" (и не разрешает его переписывать через свойство headers), чтобы уведомить веб-сервер о том, что запрос поступает из до-

кумента с другим источником. Если сервер отвечает на запрос с надлежащим заголовком "Access-Control-Allow-Origin", тогда запрос продолжится. В противном случае, если сервер явно не разрешил запрос, то объект Promise, возвращенный fetch(), отклоняется.

Прекращение запроса

Иногда возникает необходимость прекратить уже выданный запрос fetch(), возможно, из-за того, что пользователь щелкнул на кнопке отмены, или оттого, что запрос выполняется слишком долго. API-интерфейс извлечения позволяет прекращать запросы с использованием классов AbortController и AbortSignal. (Указанные классы определяют обобщенный механизм прекращения, подходящий для применения также и другими API-интерфейсами.)

Если вы хотите иметь возможность прекращения запроса fetch(), тогда создайте объект контроллера AbortController перед запуском запроса. Свойство signal объекта контроллера представляет собой объект сигнала AbortSignal. Передайте этот объект сигнала в качестве значения свойства signal объекта Options, который вы передаете функции fetch(). Теперь вы можете вызвать метод abort() объекта контроллера, чтобы прекратить запрос, что приведет к отклонению любых объектов Promise, связанных с запросом на извлечение, и выдаче исключения.

Ниже приведен пример использования механизма AbortController для обеспечения тайм-аута в отношении запросов на извлечение:

```
// Эта функция похожа на fetch(), но добавляет поддержку свойства
// тайм-аута (timeout) в объекте параметров (options) и прекращает
// извлечение, если оно не завершилось за количество миллисекунд,
// указанное в timeout.
function fetchWithTimeout(url, options={}) {
    if (options.timeout) {  // Если свойство timeout существует
                            // и не равно нулю,
        let controller = new AbortController();    // тогда создать
                                                   // контроллер
        options.signal = controller.signal;        // и установить
                                                   // свойство signal.
        // Запустить таймер, который будет посылать сигнал прекращения
        // по прошествии указанного количество миллисекунд. Обратите
        // внимание, что мы никогда не отменяем этот таймер. Вызов abort()
        // после того, как извлечение завершено, не имеет эффекта.
        setTimeout(() => { controller.abort(); }, options.timeout);
    }
    // Теперь просто выполнить нормальное извлечение.
    return fetch(url, options);
}
```

Смешанные параметры запроса

Вы видели, что функции fetch() во втором аргументе (или конструктору Request() тоже во втором аргументе) можно передавать объект Options для указания метода запроса, заголовков запроса и тела запроса.

Поддерживаются и другие параметры, включая перечисленные ниже.

- **cache**. Это свойство применяется для того, чтобы переопределить стандартное поведение кеширования браузера. Кеширование HTTP — сложная тема, которая выходит за рамки настоящей книги, но если вам известно что-то о его работе, тогда вы можете использовать описанные далее допустимые значения cache.
  - "default". Это значение задает стандартное поведение кеширования. Свежие ответы в кеше используются прямо из кеша, а давние ответы повторно проверяются перед использованием.
  - "no-store". Это значение заставляет браузер игнорировать свой кеш. Кеш не проверяется на предмет совпадений, когда делается запрос, и не обновляется, когда поступает ответ.
  - "reload". Это значение сообщает браузеру о необходимости всегда делать нормальный сетевой запрос, игнорируя кеш. Однако когда поступает ответ, он сохраняется в кеше.
  - "no-cache". Это (неудачно именованное) значение указывает браузеру на то, чтобы он не использовал свежие значения из кеша. Свежие или давние кешированные значения перед возвращением повторно проверяются.
  - "force-cache". Это значение сообщает браузеру о необходимости использования ответов из кеша, даже если они давние.
- **redirect**. Это свойство управляет тем, как браузер обрабатывает ответы перенаправления от сервера. Три допустимых значения описаны ниже.
  - "follow". Это стандартное значение, которое заставляет браузер автоматически следовать перенаправлениям. В случае его применения объекты Response, получаемые из fetch(), никогда не должны иметь значение свойства status в диапазоне между 300 и 399.
  - "error". Это значение заставляет функцию fetch() отклонять возвращаемый ею объект Promise, если сервер возвращает ответ перенаправления.
  - "manual". Это значение говорит о том, что вы хотите вручную обрабатывать ответы перенаправления, и объект Promise, возвращаемый функцией fetch(), может быть разрешен в объект Response со значением свойства status в диапазоне между 300 и 399. В таком случае вам придется использовать заголовок "Location" объекта Response, чтобы вручную следовать перенаправлению.
- **referrer**. Вы можете установить это свойство в строку, содержащую относительный URL, чтобы указать значение HTTP-заголовка "Referer" (который исторически имеет неправильное написание с тремя буквами R вместо четырех). Если вы установите данное свойство в пустую строку, тогда заголовок "Referer" не будет входить в состав запроса.

15.11.2. События, посылаемые сервером

Фундаментальная особенность протокола HTTP, с учетом которой построена веб-сеть, связана с тем, что клиенты инициируют запросы, а серверы отвечают на эти запросы. Тем не менее, некоторые веб-приложения считают полезным, чтобы их сервер посылал уведомления, когда происходят события. Это не является естественным для HTTP, но разработанная методика заключается в том, что клиент делает запрос к серверу, и затем ни клиент, ни сервер не закрывают подключение. Когда серверу есть о чем сообщить клиенту, он записывает данные в подключение, но оставляет его открытым. Результат оказывается таким, как будто клиент делает сетевой запрос, а сервер отвечает медленным и прерывистым образом со значительными паузами между всплесками активности. Сетевые подключения подобного рода обычно не остаются открытыми навсегда, но если клиент обнаруживает, что подключение закрыто, он может просто сделать еще один запрос, чтобы повторно открыть подключение.

Описанная выше методика, позволяющая серверам посылать сообщения клиентам, удивительно эффективна (хотя может быть затратной на стороне сервера, потому что сервер должен поддерживать активные подключения для всех своих клиентов). Поскольку данная методика представляет собой полезный программный шаблон, в JavaScript стороны клиента она реализована с помощью API-интерфейса EventSource. Чтобы создать такое долговременное подключение к веб-серверу, нужно просто передать URL конструктору `EventSource()`. Когда сервер записывает (надлежащим образом сформатированные) данные в подключение, объект `EventSource` транслирует их в события, которые вы можете прослушивать:

```
let ticker = new EventSource("stockprices.php");
ticker.addEventListener("bid", (event) => {
   displayNewBid(event.data);
});
```

Объект события, ассоциированный с сообщением, имеет свойство `data`, которое хранит любую строку, отправленную сервером, как полезную нагрузку этого события. Подобно всем объектам событий данный объект события также имеет свойство `type`, которое указывает имя события. Тип генерируемых событий устанавливает сервер. Если сервер опускает имя события в записываемых им данных, тогда по умолчанию типом события становится `"message"`.

Протокол SSE прямолинеен. Клиент инициирует подключение к серверу (создавая объект `EventSource`) и сервер сохраняет это подключение открытым. Когда происходит какое-то событие, сервер записывает строки текста в подключение. Событие, передаваемое по сети, может выглядеть так, не учитывая комментарии:

```
event: bid      // Устанавливает тип объекта события.
data: GOOG      // Устанавливает свойство data.
data: 999       // Добавляет разделитель строк и дополнительные данные.
                // Пустая строка помечает конец события.
```

В протоколе есть ряд дополнительных деталей, которые позволяют назначать событиям заданные идентификаторы и дают возможность повторно подключающемуся клиенту сообщить серверу, каким был идентификатор в последнем полученном событии, чтобы сервер мог заново отправить любые пропущенные события. Однако такие детали на стороне клиента не видны и потому здесь не обсуждаются.

Одним очевидным приложением для SSE является обеспечение многопользовательской совместной работы вроде онлайнового чата. Чат-клиент может применять функцию `fetch()` для отправки сообщений в дискуссионную группу и подписаться на поток данных от участников группы с помощью объекта `EventSource`. В примере 15.11 демонстрируется, насколько легко написать чат-клиент такого рода посредством `EventSource`.

Пример 15.11. Реализация простого чат-клиента с использованием `EventSource`

```
<html>
<head><title>SSE Chat</title></head>
<body>
<!-- Пользовательский интерфейс чата сводится всего лишь
     к простому полю ввода текста. -->
<!-- Новые сообщения в чате будут вставляться перед этим полем ввода. -->
<input id="input" style="width:100%; padding:10px; border:solid black 2px"/>
<script>
// Позаботиться о некоторых деталях пользовательского интерфейса.
let nick = prompt("Enter your nickname"); // Получить прозвище пользователя.
let input = document.getElementById("input");  // Найти поле ввода.
input.focus();                                 // Установить клавиатурный фокус.

// Зарегистрировать прослушиватель для уведомления о новых сообщениях,
// используя EventSource.
let chat = new EventSource("/chat");
chat.addEventListener("chat", event => { // Когда поступает сообщение чата.
   let div = document.createElement("div");    // Создать элемент <div>.
   div.append(event.data);                     // Добавить текст из сообщения.
   input.before(div);                          // Добавить div перед input.
   input.scrollIntoView();      // Гарантировать видимость элемента input.
});

// Отправить сообщения пользователя серверу, используя fetch().
input.addEventListener("change", ()=>{ // Когда пользователь нажимает <Return>.
   fetch("/chat", {                    // Запустить HTTP-запрос для этого URL.
      method: "POST",                  // Сделать его запросом POST с телом,
      body: nick + ": " + input.value  // установленным в прозвище
                                       // и ввод пользователя.
   })
   .catch(e => console.error);  // Игнорировать ответ, но отображать
                                // любые ошибки.
   input.value = "";            // Очистить поле ввода.
});
</script>
</body>
</html>
```

Код стороны сервера для такой программы чата не намного сложнее кода стороны клиента. В примере 15.12 реализован простой HTTP-сервер Node. Когда клиент запрашивает корневой URL вида /, он посылает код чат-клиента, показанный в примере 15.11. Когда клиент делает запрос GET для URL вида /chat, он сохраняет объект ответа и оставляет это подключение открытым. И когда клиент делает запрос POST для /chat, он использует тело запроса как сообщение чата и записывает его с применением формата "text/event-stream" к каждому сохраненному объекту ответа. Код сервера прослушивает порт 8080, а потому после его запуска посредством Node направьте свой браузер на http://localhost:8080, чтобы подключиться и начать чат с самим собой.

Пример 15.12. Реализация чат-сервера с помощью SSE

```
// Это код JavaScript стороны сервера, рассчитанный на запуск с помощью NodeJS.
// Он реализует очень простую, полностью анонимную дискуссионную группу.
// Посредством запросов POST для /chat он отправляет новые сообщения или
// с помощью запросов GET к тому же самому URL получает сообщения в формате
// "text/event-stream".
// Запрос GET к / возвращает простой HTML-файл, который содержит
// пользовательский интерфейс стороны клиента для чата.
const http = require("http");
const fs = require("fs");
const url = require("url");

// HTML-файл для чат-клиента. Используется ниже.
const clientHTML = fs.readFileSync("chatClient.html");

// Массив объектов ServerResponse, которым мы собираемся посылать события.
let clients = [];

// Создать новый сервер и прослушивать порт 8080.
// Для его использования подключитесь к http://localhost:8080/.
let server = new http.Server();
server.listen(8080);

// Когда сервер получает новый запрос, выполнить эту функцию.
server.on("request", (request, response) => {
    // Разобрать запрошенный URL.
    let pathname = url.parse(request.url).pathname;

    // Если запрос был для "/", тогда отправить пользовательский
    // интерфейс клиентской стороны для чата.
    if (pathname === "/") { // A request for the chat UI
      response.writeHead(200, {"Content-Type": "text/html"}).end(clientHTML)
    }
    // В противном случае отправить ошибку 404 для любого пути кроме "/chat"
    // или для любого метода, отличающегося от "GET" и "POST".
    else if (pathname !== "/chat" ||
             (request.method !== "GET" && request.method !== "POST")) {
      response.writeHead(404).end();
    }
    // Если запросом для /chat был GET, тогда клиент подключается.
    else if (request.method === "GET") {
      acceptNewClient(request, response);
    }
```

```
    // Иначе запросом к /chat является POST для нового сообщения.
    else {
        broadcastNewMessage(request, response);
    }
});
// Эта функция обрабатывает запросы GET для конечной точки /chat,
// которая генерируется, когда клиент создает новый объект EventSource
// (или когда EventSource автоматически повторно подключается).
function acceptNewClient(request, response) {
    // Запомнить объект ответа, чтобы ему можно было посылать
    // будущие сообщения.
    clients.push(response);

    // Если клиент закрыл подключение, тогда удалить соответствующий
    // объект ответа из массива активных клиентов.
    request.connection.on("end", () => {
        clients.splice(clients.indexOf(response), 1);
        response.end();
    });

    // Установить заголовки и отправить начальное событие чата
    // для этого одного клиента.
    response.writeHead(200, {
        "Content-Type": "text/event-stream",
        "Connection": "keep-alive",
        "Cache-Control": "no-cache"
    });
    response.write("event: chat\ndata: Connected\n\n");

    // Обратите внимание, что мы намеренно не вызываем здесь response.end().
    // Именно сохранение подключения открытым обеспечивает работу SSE.
}

// Эта функция вызывается в ответ на запросы POST к конечной точке /chat,
// которые клиенты посылают, когда пользователи вводят новые сообщения.
async function broadcastNewMessage(request, response) {
    // Прочитать тело запроса, чтобы получить сообщение пользователя.
    request.setEncoding("utf8");
    let body = "";
    for await (let chunk of request) {
        body += chunk;
    }

    // После того, как тело прочитано, отправить пустой ответ
    // и закрыть подключение.
    response.writeHead(200).end();

    // Сформатировать сообщение в формате text/event-stream, снабжая
    // каждую строку префиксом "data: ".
    let message = "data: " + body.replace("\n", "\ndata: ");

    // Предоставить данным сообщения префикс, который определяет
    // их как событие "chat", и снабдить их суффиксом в виде двух
    // символов новой строки, которые помечают конец события.
    let event = `event: chat\n${message}\n\n`;

    // Теперь отправить это событие всем прослушивающим клиентам.
    clients.forEach(client => client.write(event));
}
```

15.11.3. Веб-сокеты

API-интерфейс WebSocket представляет собой простой интерфейс к сложному и мощному сетевому протоколу. Веб-сокеты позволяют коду JavaScript в браузере легко обмениваться текстовыми и двоичными сообщениями с сервером. Как в случае с событиями, посылаемыми сервером (SSE), клиент обязан установить подключение, но после того, как подключение установлено, сервер может асинхронно посылать сообщения клиенту. В отличие от SSE поддерживаются двоичные сообщения, и сообщения могут отправляться в обоих направлениях, а не только от сервера до клиента.

Сетевой протокол, который запускает в работу веб-сокеты, является своего рода расширением HTTP. Хотя API-интерфейс WebSocket напоминает низкоуровневые сетевые сокеты, конечные точки подключения не идентифицируются через IP-адрес и порт. Взамен при желании подключиться к службе с применением протокола WebSocket вы указываете службу с помощью URL, как делали бы для веб-службы. Тем не менее, URL веб-сокетов начинаются с `wss://`, а не `https://`. (Браузеры обычно ограничивают работу веб-сокетов только на страницах, загружаемых посредством защищенных подключений `https://`).

Чтобы установить подключение WebSocket, браузер сначала устанавливает подключение HTTP и посылает серверу заголовок `Upgrade: websocket`, запрашивающий переключение с протокола HTTP на протокол WebSocket. Это означает, что для использования веб-сокетов в коде JavaScript стороны клиента вам нужно будет работать с веб-сервером, который также воспринимает протокол WebSocket, и иметь код стороны сервера, написанный для отправки и получения данных с применением протокола WebSocket. Если ваш сервер настроен подобным образом, то далее объясняется все, что вам необходимо знать для обеспечения клиентского конца подключения. Если же ваш сервер не поддерживает протокол WebSocket, тогда рассмотрите возможность использования событий, посылаемых сервером (см. подраздел 15.11.2).

Создание, подключение и отключение от веб-сокетов

Если вы хотите обмениваться информацией с сервером, поддерживающим WebSocket, тогда создайте объект `WebSocket`, указав URL вида `wss://`, который идентифицирует желаемый сервер и службу:

```
let socket = new WebSocket("wss://example.com/stockticker");
```

Когда вы создаете объект `WebSocket`, процесс подключения начинается автоматически. Но вновь созданный объект `WebSocket` не будет подключаться при первом возвращении.

Свойство `readyState` сокета указывает, в каком состоянии находится подключение, и может иметь описанные ниже значения.

- `WebSocket.CONNECTING`. Веб-сокет подключается.
- `WebSocket.OPEN`. Веб-сокет подключен и готов к обмену данными.
- `WebSocket.CLOSING`. Подключение веб-сокета закрывается.

- `WebSocket.CLOSED`. Подключение веб-сокета закрыто; дальнейший обмен данными невозможен. Такое состояние также возникает, когда начальная попытка подключения терпит неудачу.

Когда веб-сокет переходит из состояния `CONNECTING` в состояние `OPEN`, он инициирует событие `"open"`, которое вы можете прослушивать, установив свойство `onopen` объекта `WebSocket` или вызвав метод `addEventListener()` для этого объекта.

Если в подключении веб-сокета возникает ошибка протокола или другая ошибка, тогда объект `WebSocket` инициирует событие `"error"`. Вы можете установить свойство `onerror`, чтобы определить обработчик, или воспользоваться методом `addEventListener()`.

Завершив работу с веб-сокетом, вы можете закрыть подключение вызовом метода `close()` объекта `WebSocket`. Когда веб-сокет переходит в состояние `CLOSED`, он инициирует событие `"close"`, которое вы можете прослушивать, установив свойство `onclose`.

Отправка сообщений через веб-сокет

Для отправки сообщения серверу на другом конце подключения веб-сокета просто вызовите метод `send()` объекта `WebSocket`. Метод `send()` ожидает единственный аргумент сообщения, который может быть строкой, объектом `Blob`, объектом `ArrayBuffer`, типизированным массивом или объектом `DataView`.

Метод `send()` буферизует указанное сообщение, подлежащее передаче, и возвращает управление до того, как сообщение будет фактически отправлено. Свойство `bufferedAmount` объекта `WebSocket` содержит количество байтов, которые были буферизованы, но еще не отправлены. (Удивительно, но веб-сокеты не инициируют какое-то событие, когда значение свойства `bufferedAmount` достигает нуля.)

Получение сообщений из веб-сокета

Для получения сообщений от сервера через веб-сокет зарегистрируйте обработчик событий `"message"`, либо установив свойство `onmessage` объекта `WebSocket`, либо вызвав метод `addEventListener()`. Объектом, ассоциированным с событием `"message"`, является экземпляр `MessageEvent` со свойством `data`, которое содержит сообщение сервера. Если сервер отправил текст в кодировке UTF-8, тогда в свойстве `event.data` будет строка, содержащая этот текст.

Если сервер посылает сообщение, которое состоит из двоичных данных, а не текста, то свойство `data` будет (по умолчанию) объектом `Blob`, представляющим такие данные. Если вы предпочитаете получать двоичные сообщения в виде объектов `ArrayBuffer`, а не `Blob`, тогда установите свойство `binaryType` объекта `WebSocket` в строку `"arraybuffer"`.

Существует несколько API-интерфейсов для веб-сети, в которых для обмена сообщениями применяются объекты `MessageEvent`. В некоторых API-интерфейсах подобного рода используется алгоритм структурированного кло-

нирования (см. врезку "Алгоритм структурированного клонирования" ранее в главе), чтобы предоставить возможность сложным структурам данных быть полезной нагрузкой сообщений. Веб-сокеты не входят в число таких API-интерфейсов: сообщения, которыми обмениваются через веб-сокет, представляют собой либо одиночную строку символов Unicode, либо одиночную строку байтов (в виде объекта `Blob` или `ArrayBuffer`).

Согласование протокола

Протокол WebSocket позволяет обмениваться текстовыми и двоичными сообщениями, но ничего не говорит о структуре или смысле этих сообщений. Приложения, в которых применяются веб-сокеты, обязаны строить собственный протокол передачи поверх такого простого механизма обмена сообщениями. Использование URL типа `wss://` помогает решить задачу: каждый URL обычно будет располагать собственными правилами, касающимися того, как должен происходить обмен сообщениями. Если вы пишете код для подключения к `wss://example.com/stockticker`, то вероятно знаете, что будете получать сообщения о курсе акций.

Однако протоколы имеют тенденцию развиваться. Если гипотетический протокол котировки акций обновляется, тогда вы можете определить новый URL и подключаться к обновленной службе как к `wss://example.com/stockticker/v2`. Тем не менее, управления версиями на основе URL не всегда достаточно. В случае сложных протоколов, развивающихся с течением времени, вы можете получить развернутые серверы, которые поддерживают множество версий протокола, и развернутые клиенты, поддерживающие разные наборы версий протокола.

Предвидя такую ситуацию, протокол WebSocket и API-интерфейс включают средство согласования протокола на уровне приложения. При вызове конструктора `WebSocket()` вы передаете в первом аргументе URL типа `wss://`, но вдобавок можете передать во втором аргументе массив строк. В этом случае вы указываете список протоколов приложения, с которыми знаете, как обращаться, и предлагаете серверу выбрать один из них. Во время процесса подключения сервер выберет один из протоколов (или потерпит неудачу с выдачей ошибки, если он не поддерживает ни один из вариантов, предлагаемых клиентом). После того, как подключение установлено, свойство `protocol` объекта `WebSocket` указывает версию протокола, выбранную сервером.

15.12. Хранилище

Веб-приложения могут применять API-интерфейсы браузера для сохранения данных локально на компьютере пользователя. Такое хранилище на стороне клиента предназначено для снабжения веб-браузера памятью. Веб-приложения могут хранить, например, пользовательские предпочтения или даже свое полное состояние, чтобы восстанавливаться в точности там, где вы их оставили в конце последнего посещения. Хранилище на стороне клиента разделяется по источнику, так что страницы из одного сайта не могут читать данные, сохра-

ненные страницами из другого сайта. Но две страницы из одного сайта могут совместно использовать хранилище и применять его в качестве механизма связи. Скажем, данные, введенные в форму на одной странице, можно отображать в таблице на другой странице. Веб-приложения имеют возможность выбирать время жизни данных, которые они сохраняют: данные могут быть сохранены временно, существуя только до закрытия окна или завершения работы браузера, или же храниться на компьютере пользователя на постоянной основе, будучи доступными через многие месяцы или годы.

Есть несколько форм хранилища на стороне клиента.

Веб-хранилище

API-интерфейс веб-хранилища (Web Storage) состоит из объектов `localStorage` и `sessionStorage`, которые по существу являются постоянными объектами, отображающими строковые ключи на строковые значения. Веб-хранилище очень легко использовать и оно подходит для хранения крупных (но не гигантских) объемов данных.

Cookie-наборы

Cookie-наборы — это старый механизм хранения на стороне клиента, который был спроектирован для применения сценариями стороны сервера. Неуклюжий API-интерфейс JavaScript позволяет работать с cookie-наборами на стороне клиента, но они трудны в использовании и подходят для хранения только небольших объемов текстовых данных. Кроме того, любые данные, сохраненные как cookie-наборы, всегда передаются серверу с каждым HTTP-запросом, даже если данные интересуют только клиент.

IndexedDB

IndexedDB представляет собой асинхронный API-интерфейс к объектной базе данных, которая поддерживает индексацию.

Хранилище, безопасность и конфиденциальность

Веб-браузеры часто предлагают запомнить для вас пароли из веб-сети и сохраняют их на устройстве безопасным образом в зашифрованной форме. Но ни одна из форм хранилища данных на стороне клиента, рассматриваемых в главе, не вовлекает шифрование: вы должны принимать во внимание, что вся сохраняемая вашими веб-приложениями информация находится на устройстве пользователя в незашифрованном виде. Следовательно, сохраненные данные доступны любопытным пользователям, которые совместно используют устройство, и существующему на устройстве вредоносному программному обеспечению (вроде шпионского ПО). По указанной причине ни одна из форм хранилища данных на стороне клиента никогда не должна применяться для хранения паролей, номеров банковских счетов или другой конфиденциальной информации.

15.12.1. localStorage и sessionStorage

Свойства localStorage и sessionStorage объекта Window ссылаются на объекты Storage. Объект Storage ведет себя во многом похоже на обыкновенный объект JavaScript за исключением описанных ниже моментов.

- Значения свойств объектов Storage обязаны быть строками.
- Свойства, сохраненные в объекте Storage, постоянны. Если вы устанавливаете какое-то свойство объекта localStorage, после чего пользователь перезагружает страницу, то значение, сохраненное вами в этом свойстве, по-прежнему будет доступно вашей программе.

Вот как можно использовать объект localStorage:

```
let name = localStorage.username;    // Запросить сохраненное значение.
if (!name) {
    name = prompt("What is your name?"); // Задать пользователю вопрос
                                         // насчет его имени.
    localStorage.username = name;        // Сохранить ответ
                                         // пользователя.
}
```

Вы можете применять операцию delete для удаления свойств из объектов localStorage и sessionStorage, а также применять цикл for/in или Object.keys() для перечисления свойств объекта Storage. Если вы хотите удалить все свойства из объекта хранилища, тогда вызовите метод clear():

```
localStorage.clear();
```

В объектах хранилища также определены методы getItem(), setItem() и deleteItem(), которые вы при желании можете использовать вместо прямого доступа к свойствам и операции delete.

Имейте в виду, что свойства объектов Storage способны хранить только строки. Если вы хотите сохранять и извлекать другие разновидности данных, то должны кодировать и декодировать их самостоятельно.

Например:

```
// Когда вы сохраняете число, оно преобразуется в строку автоматически.
// Не забывайте разбирать его после извлечения из хранилища.
localStorage.x = 10;
let x = parseInt(localStorage.x);

// Преобразовать объект Date в строку при установке
// и разобрать его при получении.
localStorage.lastRead = (new Date()).toUTCString();
let lastRead = new Date(Date.parse(localStorage.lastRead));

// JSON обеспечивает удобное кодирование для любых
// элементарных типов или структур данных.
localStorage.data = JSON.stringify(data); //Закодировать и сохранить.
let data = JSON.parse(localStorage.data); // Извлечь и декодировать.
```

Время жизни и границы хранилища

Отличие между объектами `localStorage` и `sessionStorage` касается времени жизни и области видимости хранилища. Данные, сохраненные через `localStorage`, постоянны: срок их хранения не истекает, и они остаются на устройстве пользователя до тех пор, пока веб-приложение не удалит их или пользователь не затребует их удаление у браузера (посредством пользовательского интерфейса, специфичного для браузера).

Объект `localStorage` ограничен источником документа. Как объяснялось в подразделе "Политика одинакового источника" ранее в главе, источник документа определяется его протоколом, именем хоста и портом. Все документы с одинаковым источником разделяют те же самые данные `localStorage` (независимо от источника сценариев, которые фактически производят доступ к `localStorage`). Они могут читать и перезаписывать данные друг друга. Но документы с разными источниками никогда не смогут читать или перезаписывать данные друг друга (даже если они выполняют сценарий из того же самого стороннего сервера).

Обратите внимание, что объект `localStorage` также ограничивается реализацией браузера. Если вы посещаете сайт с применением Firefox и затем снова посещаете его, используя Chrome (например), тогда любые данные, сохраненные во время первого посещения, не будут доступны при втором посещении.

Данные, сохраненные через объект `sessionStorage`, имеют время жизни, отличающееся от времени жизни данных, сохраненных через `localStorage`: время жизни у них такое же, как у окна верхнего уровня или вкладки браузера, где выполняется сценарий, который их сохраняет. Когда окно или вкладка закрывается навсегда, то любые данные, сохраненные посредством `sessionStorage`, удаляются. (Однако важно отметить, что современные браузеры обладают возможностью повторно открыть недавно закрытые вкладки и восстановить последние сеансы просмотра, поэтому время жизни таких вкладок и ассоциированных с ними объектов `sessionStorage` может быть дольше, чем кажется.)

Подобно `localStorage` объект `sessionStorage` ограничен источником документа, так что документы с разными источниками никогда не будут разделять `sessionStorage`. Но `sessionStorage` также ограничивается на базе окон. Если у пользователя открыты две вкладки браузера, отображающие документы из того же самого источника, то эти две вкладки имеют отдельные данные `sessionStorage`: сценарии, выполняющиеся в одной вкладке, не могут читать или перезаписывать данные, записанные сценариями в другой вкладке, даже когда обе вкладки посещают в точности ту же страницу и выполняют те же самые сценарии.

События хранилища

Всякий раз, когда хранящиеся в `localStorage` данные изменяются, браузер инициирует событие `"storage"` в любом другом объекте `Window`, где изменяемые данные видны (но не в окне, которое вносит изменение). Если в браузере открыты две вкладки, отображающие страницы с одинаковым источником, и одна из страниц сохраняет значение в `localStorage`, то другая вкладка получит событие `"storage"`.

Обработчик для событий "storage" регистрируется либо установкой свойства window.onstorage, либо вызовом метода window.addEventListener() с типом события "storage".

Объект события, ассоциированный с событием "storage", имеет ряд важных свойств.

- **key**. Имя или ключ элемента, который был установлен или удален. Если вызывался метод clear(), тогда это свойство будет равно null.
- **newValue**. Содержит в себе новое значение элемента при его наличии. Если вызывался метод removeItem(), тогда это свойство не будет присутствовать.
- **oldValue**. Содержит в себе старое значение существующего элемента, который был изменен или удален. Если добавляется новое свойство (без старого значения), тогда это свойство не будет присутствовать в объекте события.
- **storageArea**. Объект Storage, который изменяется. Обычно это объект localStorage.
- **url**. URL (в виде строки) документа, чей сценарий изменил это хранилище.

Обратите внимание, что объект localStorage и событие "storage" могут служить механизмом широковещательной рассылки, с помощью которого браузер отправляет событие всем окнам, в текущий момент посещающим тот же самый веб-сайт. Скажем, если пользователь требует у веб-сайта прекратить выполнение анимации, тогда сайт мог бы сохранить такое предпочтение в localStorage, чтобы соблюсти его при будущих посещениях. И сохраняя предпочтение, он генерирует событие, которое позволяет остальным окнам, отображающим тот же сайт, также соблюсти такое требование.

В качестве еще одного примера вообразим себе веб-приложение для редактирования изображений, которое позволяет пользователю отображать палитры инструментов в отдельных окнах. Когда пользователь выбирает инструмент, веб-приложение применяет localStorage для сохранения текущего состояния и отправки другим окнам уведомления о том, что был выбран новый инструмент.

15.12.2. Cookie-наборы

Cookie-набор — это небольшой объем именованных данных, который сохранен веб-браузером и ассоциирован с определенной веб-страницей или веб-сайтом. Cookie-наборы проектировались для программирования на стороне сервера и на самом низком уровне они реализованы как расширение протокола HTTP. Данные cookie-наборов автоматически передаются между веб-браузером и веб-сервером, так что сценарии стороны сервера могут читать и записывать значения cookie-наборов, которые хранятся на стороне клиента. В настоящем подразделе будет показано, как манипулировать cookie-наборами также в сценариях стороны клиента с использованием свойства cookie объекта Document.

Почему они называются "cookie-наборами"?

Название "cookie-набор" не обладает большим смыслом, но и не лишено предыстории. В анналах истории компьютеров термин "cookie-набор" или "магический cookie-набор" применялся для обозначения небольшой порции данных, в частности фрагмента привилегированных или секретных данных сродни паролю, который подтверждает идентичность или разрешает доступ. В JavaScript cookie-наборы используются для сохранения состояния и могут устанавливать своего рода идентичность для веб-браузера. Тем не менее, cookie-наборы в JavaScript не задействуют криптографию и никак не защищены (хотя помогает их передача через подключение `https:`).

API-интерфейс для манипулирования cookie-наборами является старым и непонятным. Какие-либо методы отсутствуют: cookie-наборы запрашиваются, устанавливаются и удаляются за счет чтения и записи свойства `cookie` объекта `Document` с применением специально сформатированных строк. Время жизни и границы каждого cookie-набора могут указываться индивидуально посредством атрибутов cookie-набора, которые тоже задаются с помощью специально сформатированных строк, устанавливаемых в том же самом свойстве `cookie`.

В последующих подразделах объясняется, как запрашивать и устанавливать значения и атрибуты cookie-наборов.

Чтение cookie-наборов

Чтение свойства `document.cookie` приводит к возвращению строки, содержащей все cookie-наборы, которые применяются к текущему документу. Строка представляет собой список пар имя/значение, отделенных друг от друга точкой с запятой и пробелом. Значение cookie-набора — это просто само значение и оно не включает какие-либо атрибуты, которые могут быть ассоциированы с данным cookie-набором. (Мы обсудим атрибуты в следующем подразделе.) Для использования свойства `document.cookie` обычно требуется вызвать метод `split()`, чтобы разбить его на индивидуальные пары имя/значение.

После того, как значение cookie-набора извлечено из свойства `cookie`, его необходимо интерпретировать на основе того формата или кодировки, которая применялась создателем cookie-набора. Скажем, можно передать значение cookie-набора методу `decodeURIComponent()` и затем `JSON.parse()`.

В приведенном ниже коде определяется функция `getCookie()`, которая производит разбор свойства `document.cookie` и возвращает объект, чьи свойства задают имена и значения cookie-наборов документа:

```
// Возвращает cookie-наборы документа как объект Map.
// Предполагает, что значения cookie-наборов
// закодированы посредством encodeURIComponent().
function getCookies() {
    let cookies = new Map();      // Объект, который будет возвращен.
    let all = document.cookie;    // Получить все cookie-наборы в одной
                                  // большой строке.
```

```
    let list = all.split("; ");          // Разбить на индивидуальные
                                         // пары имя/значение.
    for(let cookie of list) {//Для каждого cookie-набора в этом списке:
      if (!cookie.includes("=")) continue;  // Пропустить,
                                            // если нет знака =.
      let p = cookie.indexOf("=");       // Найти первый знак =.
      let name = cookie.substring(0, p);// Получить имя cookie-набора
      let value = cookie.substring(p+1);// Получить значение
                                         // cookie-набора.
      value = decodeURIComponent(value);// Декодировать значение.
      cookies.set(name, value);          // Запомнить имя и значение
                                         // cookie-набора.
    }
    return cookies;
}
```

Атрибуты cookie-наборов: время жизни и границы

Помимо имени и значения каждый cookie-набор имеет необязательные атрибуты, которые управляют его временем жизни и границами. Прежде чем приступить к установке cookie-наборов в коде JavaScript, нужно объяснить атрибуты cookie-наборов.

По умолчанию cookie-наборы являются временными; хранящиеся в них значения удерживаются на протяжении сеанса веб-браузера, но утрачиваются, когда пользователь завершает работу браузера. Если вы хотите, чтобы cookie-набор сохранился после одиночного сеанса просмотра, тогда должны сообщить браузеру, насколько долго (в секундах) желательно удерживать cookie-набор, указав атрибут `max-age`. Если вы задаете время жизни, то браузер будет сохранять cookie-наборы в файле и удалять их только после истечения срока их действия.

Видимость cookie-наборов ограничена не только источником документа как у объектов `localStorage` и `sessionStorage`, но и путем к документу. Границы допускают конфигурирование через атрибуты `path` и `domain` cookie-набора. По умолчанию cookie-набор ассоциирован и доступен веб-странице, которая его создала, и любым другим веб-страницам в том же каталоге или любых подкаталогах этого каталога. Например, если веб-страница `example.com/catalog/index.html` создает cookie-набор, то данный cookie-набор также будет видимым веб-страницам `example.com/catalog/order.html` и `example.com/catalog/widgets/index.html`, но не веб-странице `example.com/about.html`.

Такое стандартное поведение видимости часто оказывается именно тем, что желательно. Однако временами значение cookie-набора необходимо использовать по всему веб-сайту вне зависимости от того, какая страница создала cookie-набор. Скажем, если пользователь вводит свой почтовый адрес внутри формы на одной странице, то вы можете сохранить введенный адрес для применения в качестве принятого по умолчанию в следующий раз, когда пользователь вернется на страницу, и также в совершенно несвязанной форме на другой странице, где пользователю предлагается ввести адрес для отправки счета. Чтобы сделать возможным подобное использование, вы указываете

атрибут path для cookie-набора. Затем любая веб-страница из того же самого веб-сервера, URL которой начинается с префикса, заданного вами в path, сможет разделять cookie-набор. Например, если cookie-набор, созданный веб-страницей example.com/catalog/widgets/index.html, имеет атрибут path, установленный в "/catalog", тогда этот cookie-набор также будет видимым веб-странице example.com/catalog/order.html. А если атрибут path установлен в "/", то cookie-набор окажется видимым любой веб-странице в домене example.com, обеспечивая cookie-набору границы как у localStorage.

По умолчанию cookie-наборы ограничиваются источником документа. Тем не менее, крупным веб-сайтам может требоваться разделение cookie-наборов между поддоменами. Скажем, у сервера в order.example.com может возникнуть необходимость в чтении значений cookie-наборов, установленных в catalog.example.com. Здесь на помощь приходит атрибут domain. Если установить атрибут path в "/" и атрибут domain в ".example.com" для cookie-набора, созданного веб-страницей в catalog.example.com, то такой cookie-набор станет доступным всем веб-страницам в catalog.example.com, orders.example.com и любым другим серверам в домене example.com. Обратите внимание, что атрибут domain для cookie-набора нельзя устанавливать в домен, отличающийся от родительского домена вашего сервера.

Последний булевский атрибут cookie-набора называется secure и указывает, каким образом значения cookie-набора передаются по сети. По умолчанию cookie-наборы не защищены, т.е. они передаются через нормальное незащищенное подключение HTTP. Однако если cookie-набор помечен как защищенный, тогда он передается, только когда браузер и сервер связываются через HTTPS или другой защищенный протокол.

Ограничения cookie-наборов

Cookie-наборы предназначены для сохранения данных небольшого объема сценариями стороны сервера, и эти данные передаются серверу каждый раз, когда запрашивается соответствующий URL. Стандарт, который определяет cookie-наборы, поощряет производителей браузеров разрешать неограниченное количество cookie-наборов неограниченного размера, но он не требует от браузеров хранения более 300 cookie-наборов в общей сложности, более 20 cookie-наборов на веб-сервер или более 4 Кбайт данных на cookie-набор (в лимите 4 Кбайт учитываются и имя, и значение). На практике браузеры разрешают гораздо больше, чем 300 cookie-наборов в сумме, но некоторые браузеры по-прежнему могут применять лимит 4 Кбайт.

Сохранение cookie-наборов

Чтобы ассоциировать значение временного cookie-набора с текущим документом, просто установите свойство cookie в строку имя=значение, например:

```
document.cookie=`version=${encodeURIComponent(document.lastModified)}`;
```

Когда вы прочитаете свойство cookie в следующий раз, сохраненная вами пара имя/значение будет включена в список cookie-наборов для документа. Значения cookie-наборов не могут содержать точки с запятой, запятые или пробельные символы. По этой причине вы можете использовать глобальную функцию encodeURIComponent() базового языка JavaScript для кодирования значений перед их сохранением в cookie-наборах. В таком случае при чтении значений cookie-наборов вам придется применять функцию decodeURIComponent().

Cookie-набор, записанный с помощью простой пары имя/значение, сохраняется для текущего сеанса просмотра, но утрачивается, когда пользователь завершает работу браузера. Чтобы создать cookie-набор, который способен сохраняться между сеансами браузера, укажите его время жизни (в секундах) посредством атрибута max-age. Вы можете делать это установкой свойства cookie в строку вида name=value; max-age=seconds. В показанной ниже функции устанавливается cookie-набор с необязательным атрибутом max-age:

```
// Сохраняет пару имя/значение как cookie-набор, кодируя значение
// с помощью encodeURIComponent() для отмены точек с запятой,
// запятых и пробелов.
// Если в daysToLive передано число, тогда функция устанавливает
// атрибут max-age, чтобы cookie-набор утратил силу через
// указанное количество дней.
// Передача 0 приводит к удалению cookie-набора.
function setCookie(name, value, daysToLive=null) {
    let cookie = `${name}=${encodeURIComponent(value)}`;
    if (daysToLive !== null) {
        cookie += `; max-age=${daysToLive*60*60*24}`;
    }
    document.cookie = cookie;
}
```

Точно так же вы можете устанавливать атрибуты path и domain для cookie-набора, присоединяя строки вида ;path=value или ;domain=value к строке, в которую устанавливаете свойство document.cookie. Для установки атрибута secure просто добавьте ;secure.

Чтобы изменить значение cookie-набора, снова установите его значение с использованием того же самого имени, пути и домена, но нового значения. При изменении значения cookie-набора вы можете изменить его время жизни, указав новый атрибут max-age.

Для удаления cookie-набора установите его снова с применением того же самого имени, пути и домена, указав произвольное (или пустое) значение и атрибут max-age с 0.

15.12.3. IndexedDB

Архитектура веб-приложений традиционно предлагала HTML, CSS и JavaScript на стороне клиента и базу данных на стороне сервера. Тем не менее, вас может удивить, что веб-платформа включает простую объектную базу данных с API-интерфейсом JavaScript для постоянного хранения объектов JavaScript на компьютере пользователя и их извлечения по мере необходимости.

IndexedDB является объектной, а не реляционной базой данных, и она гораздо проще баз данных, которые поддерживают запросы SQL. Однако она мощнее, эффективнее и надежнее, чем хранилище ключей/значений, предлагаемое объектом `localStorage`. Подобно `localStorage` базы данных IndexedDB ограничены источником содержащего документа: две веб-страницы с одинаковым источником могут иметь доступ к данным друг друга, но веб-страницы из отличающихся источников — не могут.

Каждый источник может иметь любое количество баз данных IndexedDB. Каждая база данных имеет имя, которое должно быть уникальным внутри источника. В API-интерфейсе IndexedDB база данных — это просто коллекция именованных *объектных хранилищ*. Как следует из названия, объектное хранилище запоминает объекты. Объекты сериализируются в объектное хранилище с использованием алгоритма структурированного клонирования (см. врезку "Алгоритм структурированного клонирования" ранее в главе), т.е. сохраняемые объекты могут иметь свойства, чьими значениями являются объекты `Map`, `Set` или типизированные массивы. Каждый объект обязан иметь ключ, по которому он может сортироваться и извлекаться из хранилища. Ключи должны быть уникальными — два объекта в том же самом хранилище не могут иметь одинаковые ключи — и они должны поддерживать естественное упорядочение, чтобы их можно было сортировать. Допустимыми ключами в JavaScript будут строки, числа и объекты `Date`. База данных IndexedDB способна автоматически генерировать уникальный ключ для каждого объекта, вставляемого в базу данных. Тем не менее, часто объекты, вставляемые вами в объектное хранилище, уже будут иметь свойство, которое подходит для применения в качестве ключа. В таком случае при создании объектного хранилища вы указываете "путь к ключу" для этого свойства. Концептуально путь к ключу представляет собой значение, которое сообщает базе данных, каким образом извлекать ключ из объекта.

Помимо извлечения объектов из объектного хранилища по значению их первичного ключа вас может интересовать возможность поиска на основе значений других свойств в объекте. Для этого в объектном хранилище можно определить любое количество *индексов*. (Наличие возможности индексации и объясняет название IndexedDB.) Каждый индекс определяет вторичный ключ для сохраненных объектов. Такие индексы обычно не уникальны и одиночному значению ключа могут соответствовать сразу несколько объектов.

IndexedDB обеспечивает гарантии атомарности: запросы и обновления базы данных группируются внутри *транзакции*, так что они все вместе успешны или все вместе отказывают и никогда не оставляют базу данных в неопределенном, частично обновленном состоянии. Транзакции в IndexedDB проще, чем во многих API-интерфейсах баз данных; позже мы снова к ним вернемся.

Концептуально API-интерфейс IndexedDB довольно прост. Чтобы запросить или обновить базу данных, сначала вы должны ее открыть (указав имя). Далее вы создаете объект транзакции и используете его для нахождения в базе данных желаемого объектного хранилища, тоже по имени. В заключение вы ищете объект, вызывая метод `get()` объектного хранилища, либо сохраняете новый объект, вызывая метод `put()` (или `add()`, если хотите избежать перезаписывания существующих объектов).

Если вы хотите найти объекты для диапазона ключей, тогда создайте объект IDBRange, который задает верхнюю и нижнюю границы диапазона, и передайте его методу getAll() или openCursor() объектного хранилища.

Если вы желаете сделать запрос с применением вторичного ключа, тогда найдите именованный индекс объектного хранилища и затем вызовите метод get(), getAll() или openCursor() объекта индекса, передавая ему либо одиночный ключ, либо объект IDBRange.

Однако концептуальная простота API-интерфейса IndexedDB осложняется тем фактом, что API-интерфейс является асинхронным (поэтому веб-приложения могут его использовать без блокирования главного потока пользовательского интерфейса браузера). API-интерфейс IndexedDB был определен до широкой поддержки объектов Promise, так что он основан на событиях, а не на Promise, т.е. не работает с ключевыми словами async и await.

Создание транзакций и поиск объектных хранилищ и индексов представляют собой синхронные операции. Но открытие базы данных, обновление объектного хранилища и запрашивание хранилища или индекса относятся к асинхронным операциям. Все эти асинхронные методы немедленно возвращают объект запроса. Браузер инициирует события успеха или неудачи в объекте запроса, когда запрос завершается успешно или отказывает, и вы можете определить обработчики с помощью свойств onsuccess и onerror. Внутри обработчика onsuccess результат операции доступен как свойство result объекта запроса. Еще одно удобное событие, "complete", отправляется объектам транзакций, когда транзакция успешно завершена.

Полезная особенность такого асинхронного API-интерфейса заключается в том, что он упрощает управление транзакциями. API-интерфейс IndexedDB вынуждает вас создавать объект транзакции для получения объектного хранилища, в котором вы можете выполнять запросы и обновления. В синхронном API-интерфейсе вы ожидали бы явной пометки конца транзакции за счет вызова метода commit(). Но в IndexedDB транзакции фиксируются автоматически (если вы их явно не прекращаете), когда все обработчики событий onsuccess были запущены и больше нет незаконченных асинхронных запросов, которые относятся к данной транзакции.

Есть еще одно событие, которое важно для API-интерфейса IndexedDB. Когда вы открываете базу данных в первый раз или инкрементируете номер версии существующей базы данных, IndexedDB инициирует событие "upgradeneeded" в объекте запроса, возвращенном вызовом indexedDB.open(). Задача обработчика событий "upgradeneeded" — определить или обновить схему для новой базы данных (или новой версии существующей базы данных). В случае баз данных IndexedDB это означает создание объектных хранилищ и определение в них индексов. Фактически API-интерфейс IndexedDB позволяет вам создать объектное хранилище либо индекс только в ответ на событие "upgradeneeded".

Ознакомившись с предложенным высокоуровневым обзором IndexedDB, вы должны быть в состоянии понять пример 15.13, в котором IndexedDB применяется для создания и запрашивания базы данных, отображающей почтовые коды США на города. В нем демонстрируются многие (но не все) основные средства IndexedDB. Код примера 15.13 длинный, но он хорошо прокомментирован.

Пример 15.13. База данных IndexedDB почтовых кодов США

```
// Эта служебная функция асинхронно получает объект базы данных
// (при необходимости создавая и инициализируя базу данных)
// и передает ее обратному вызову.
function withDB(callback) {
   let request = indexedDB.open("zipcodes", 1);   // Запросить версию 1
                                                  // базы данных.
   request.onerror = console.error;  // Сообщать о любых ошибках
   request.onsuccess = () => {  // или вызывать это в случае успеха.
      let db = request.result;  // Результатом запроса будет база данных.
      callback(db);             // Вызвать обратный вызов с базой данных.
   };
   // Если версия 1 базы данных пока не существует, тогда будет запущен
   // этот обработчик событий. Он используется для создания и инициализации
   // объектных хранилищ и индексов, когда база данных впервые создается,
   // или для их модификации, когда мы переключаемся с одной версии схемы
   // базы данных на другую.
   request.onupgradeneeded = () => { initdb(request.result, callback); };
}
// withDB() вызывает эту функцию, если база данных пока еще
// не инициализирована. Мы устанавливаем базу данных и наполняем ее данными,
// затем передаем базу данных функции обратного вызова.
//
// Наша база данных почтовых кодов включает одно объектное хранилище,
// которое содержит объекты следующего вида:
//
//   {
//     zipcode: "02134",
//     city: "Allston",
//     state: "MA",
//   }
//
// Мы используем свойство zipcode в качестве ключа базы данных
// и создаем индекс для названия города (свойства city).
function initdb(db, callback) {
   // Создать объектное хранилище, задав имя хранилища и объект
   // параметров, который включает "путь к ключу", указывающий
   // имя свойства поля ключа для этого хранилища.
   let store = db.createObjectStore("zipcodes",  // Имя хранилища.
                                    { keyPath: "zipcode" });
   // Индексировать объектное хранилище по названию города и почтовому коду.
   // В этом методе строка с путем к ключу передается непосредственно как
   // обязательный аргумент, а не как часть объекта параметров.
   store.createIndex("cities", "city");
   // Получить данные, которыми мы собираемся инициализировать базу данных.
   // Файл данных zipcodes.json был сгенерирован из данных, подпадающих
   // под лицензию Creative Commons и доступных в www.geonames.org:
   // https://download.geonames.org/export/zip/US.zip.
   fetch("zipcodes.json")                    // Сделать HTTP-запрос GET.
      .then(response => response.json())     // Разобрать тело как JSON.
```

```
    .then(zipcodes => {                          // Получить 40 Кбайт записей
                                                 // с почтовыми кодами.
        // Для вставки данных почтовых кодов в базу данных нам необходим
        // объект транзакции. Чтобы создать объект транзакции, нужно
        // указать, какие объектные хранилища мы будем использовать
        // (у нас есть только одно), и сообщить ему, что мы планируем
        // выполнять запись в базу данных, а не только чтение:
        let transaction = db.transaction(["zipcodes"], "readwrite");
        transaction.onerror = console.error;

        // Получить наше объектное хранилище из транзакции.
        let store = transaction.objectStore("zipcodes");

        // Наилучшая характеристика API-интерфейса IndexedDB заключается
        // в том, что объектные хранилища *действительно* просты.
        // Вот как можно добавлять (или обновлять) наши записи:
        for(let record of zipcodes) { store.put(record); }

        // После успешного завершения транзакции база данных инициализи-
        // рована и готова к использованию, так что мы можем вызвать функцию
        // обратного вызова, которая первоначально была передана withDB().
        transaction.oncomplete = () => { callback(db); };
    });
}

// Для заданного почтового кода функция использует API-интерфейс IndexedDB,
// чтобы асинхронно искать название города с таким почтовым кодом,
// и передает его указанному обратному вызову или null, если город не найден.
function lookupCity(zip, callback) {
    withDB(db => {
        // Создать для этого запроса объект транзакции, выполняющей только
        // чтение. Аргументом является массив объектных хранилищ,
        // которые необходимо использовать.
        let transaction = db.transaction(["zipcodes"]);

        // Получить объектное хранилище из транзакции.
        let zipcodes = transaction.objectStore("zipcodes");

        //Запросить объект,который соответствует указанному ключу почтового кода.
        // Строки кода выше были синхронными, но следующие строки асинхронные.
        let request = zipcodes.get(zip);
        request.onerror = console.error;  // Сообщать о любых ошибках
        request.onsuccess = () => {       // или вызывать это в случае успеха.
            let record = request.result;  // Это результат запроса.
            if (record) {     // Если совпадение найдено, тогда передать
                              // его обратному вызову.
                callback(`${record.city}, ${record.state}`);
            } else {          // Иначе сообщить обратному вызову о том,
                              // что ничего не найдено.
                callback(null);
            }
        };
    });
}
```

```
// Для заданного названия города функция использует API-интерфейс IndexedDB,
// чтобы асинхронно искать все записи с почтовыми кодами для всех городов
// (в любых штатах), которые имеют такое (чувствительное к регистру) название.
function lookupZipcodes(city, callback) {
  withDB(db => {
    // Как и ранее, мы создаем транзакцию и получаем объектное хранилище.
    let transaction = db.transaction(["zipcodes"]);
    let store = transaction.objectStore("zipcodes");

    // На этот раз мы также получаем индекс городов в объектном хранилище.
    let index = store.index("cities");

    // Запросить все совпадающие записи в индексе с указанным названием
    // города и после получения передать их обратному вызову.
    // В случае ожидание большего количества результатов
    // взамен можно использовать openCursor().
    let request = index.getAll(city);
    request.onerror = console.error;
    request.onsuccess = () => { callback(request.result); };
  });
}
```

15.13. Потоки воркеров и обмен сообщениями

Одна из фундаментальных особенностей среды JavaScript заключается в том, что она однопоточная: браузер никогда не будет выполнять два обработчика событий одновременно, и не будет запускать таймер, пока выполняется обработчик событий, например. Параллельные обновления состояния приложения попросту невозможны, и программистам стороны клиента не нужно думать или даже понимать параллельное программирование. Следствием является то, что функции JavaScript стороны клиента не должны выполняться слишком долго; в противном случае они будут мешать работе цикла обработки событий и веб-браузер перестанет реагировать на пользовательский ввод. Именно по этой причине, скажем, функция `fetch()` реализована как асинхронная.

Веб-браузеры очень осторожно ослабляют требование однопоточности посредством класса `Worker`: экземпляры данного класса представляют потоки, которые выполняются параллельно с главным потоком и циклом обработки событий. Воркеры находятся в автономной исполняющей среде с совершенно независимым глобальным объектом и не имеют доступа к объектам `Window` или `Document`. Воркеры могут взаимодействовать с главным потоком только посредством асинхронной передачи сообщений. В итоге параллельные изменения дерева DOM остаются невозможными, но вы можете писать длительно выполняющиеся функции, которые не останавливают цикл обработки событий и не подвешивают браузер. Создание нового воркера не является настолько тяжеловесной операцией, как открытие нового окна браузера, но воркеры — не наилегчайшие "нити", и не имеет смысла создавать новые воркеры для выполнения тривиальных операций. Для сложных веб-приложений может обнаружиться, что удобно создать десять воркеров, но вряд ли приложение с сотнями или тысячами воркеров окажется практичным.

Воркеры полезны, когда вашему приложению необходимо выполнять задачи с большим объемом вычислений, такие как обработка изображений. Использование воркеров выносит задачи подобного рода за пределы главного потока, чтобы браузер не переставал реагировать на действия пользователя. К тому же воркеры предлагают возможность распределения работы между множеством потоков. Но воркеры также удобны, когда вам нужно часто выполнять умеренно интенсивные вычисления. Предположим, например, что вы реализуете простой редактор кода в браузере и хотите обеспечить цветовое выделение синтаксиса. Чтобы добиться правильного выделения, вам необходимо синтаксически анализировать код при каждом нажатии клавиши. Но если вы сделаете это в главном потоке, то код синтаксического анализа, по всей видимости, будет препятствовать быстрому запуску обработчиков событий, которые реагируют на нажатия клавиш пользователем, и набор текста окажется замедленным.

Подобно любому потоковому API-интерфейсу API-интерфейс Worker имеет две части. Первая часть — объект `Worker`: именно так воркер выглядит снаружи для потока, который его создает. Вторая часть — `WorkerGlobalScope`: глобальный объект для нового воркера, и это то, как поток воркера выглядит внутри себя самого.

В последующих подразделах раскрываются `Worker` и `WorkerGlobalScope`, а также объясняется API-интерфейс передачи сообщений, который позволяет воркерам взаимодействовать с главным потоком и друг с другом. Тот же самый API-интерфейс связи применяется для обмена сообщениями между документом и элементами `<iframe>`, содержащимися в документе, что также обсуждается в последующих подразделах.

15.13.1. Объекты воркеров

Для создания нового воркера вызовите конструктор `Worker()`, передав ему URL, который задает код JavaScript, предназначенный для выполнения воркером:

```
let dataCruncher = new Worker("utils/cruncher.js");
```

Если вы укажете относительный URL, тогда он распознается относительно URL документа, который содержит сценарий, вызывающий конструктор `Worker()`. Если вы укажете абсолютный URL, то он должен иметь тот же самый источник (протокол, хост и порт), что и содержащий документ.

Имея объект `Worker`, вы можете посылать ему данные с помощью метода `postMessage()`. Значение, передаваемое `postMessage()`, копируется с использованием алгоритма структурированного клонирования (см. врезку "Алгоритм структурированного клонирования" ранее в главе), а результирующая копия будет доставлена воркеру через событие `"message"`:

```
dataCruncher.postMessage("/api/data/to/crunch");
```

Здесь мы просто передаем единственное строковое сообщение, но допускается также передавать объекты, массивы, типизированные массивы, объекты `Map`, `Set` и т.д. Вы можете принимать сообщения от воркера, организовав прослушивание событий `"message"` в объекте `Worker`:

```
dataCruncher.onmessage = function(e) {
    let stats = e.data;  // Сообщение - это свойство data события.
    console.log(`Average: ${stats.mean}`);
}
```

Как и все цели событий, объекты `Worker` определяют стандартные методы `addEventListener()` и `removeEventListener()`, которые можно применять вместо свойства `onmessage`.

Помимо `postMessage()` объекты `Worker` имеют еще один метод, `terminate()`, который вынуждает воркер остановить выполнение.

15.13.2. Глобальный объект в воркерах

Когда вы создаете новый воркер с помощью конструктора `Worker()`, то указываете URL файла с кодом JavaScript. Этот код выполняется в новой, чистой исполняющей среде JavaScript, изолированной от сценария, который создал воркер. Глобальным объектом в новой исполняющей среде является `WorkerGlobalScope`. Объект `WorkerGlobalScope` — нечто большее, нежели глобальный объект базового JavaScript, но меньшее, чем полноценный объект `Window` стороны клиента.

Объект `WorkerGlobalScope` имеет метод `postMessage()` и свойство обработчика событий `onmessage`, которые аналогичны таковым в объекте `Worker`, но работают в противоположном направлении: вызов `postMessage()` внутри воркера генерирует событие `"message"` снаружи воркера, а сообщения, отправленные снаружи воркера, превращаются в события и доставляются обработчику `onmessage`. Поскольку `WorkerGlobalScope` — глобальный объект в воркере, `postMessage()` и `onmessage` выглядят для кода воркера как глобальная функция и глобальная переменная.

Если вы передаете во втором аргументе конструктора `Worker()` объект и он имеет свойство `name`, тогда значения этого свойства становится значением свойства `name` в глобальном объекте воркера. Воркер может включать такое имя в любые сообщения, которые он выводит посредством `console.warn()` или `console.error()`.

Функция `close()` позволяет воркеру завершить свою работу, и по своему действию она аналогична методу `terminate()` объекта `Worker`.

Из-за того, что `WorkerGlobalScope` является глобальным объектом для воркеров, он располагает всеми свойствами глобального объекта базового JavaScript, такими как объект `JSON`, функция `isNaN()` и конструктор `Date()`. Тем не менее, вдобавок объект `WorkerGlobalScope` также имеет следующие свойства объекта `Window` стороны клиента.

- Ссылка `self` на сам глобальный объект. `WorkerGlobalScope` — не объект `Window` и в нем не определено свойство `window`.
- Методы таймера `setTimeout()`, `clearTimeout()`, `setInterval()` и `clearInterval()`.

- Свойство location, описывающее URL, который был передан конструктору Worker(). Свойство location ссылается на объект Location в точности как свойство location объекта Window. Объект Location имеет свойства href, protocol, host, hostname, port, pathname, search и hash. Однако указанные свойства в воркере допускают только чтение.
- Свойство navigator, которое ссылается на объект со свойствами, похожими на свойства объекта Navigator окна. Объект Navigator воркера имеет свойства appName, appVersion, platform, userAgent и onLine.
- Обычные методы цели события addEventListener() и removeEventListener().

Наконец, в состав объекта WorkerGlobalScope входят важные API-интерфейсы JavaScript стороны клиента, включая объект Console, функцию fetch() и API-интерфейс IndexedDB. Объект WorkerGlobalScope также имеет конструктор Worker(), т.е. потоки воркеров могут создавать собственные воркеры.

15.13.3. Импортирование кода в воркер

Воркеры были определены в веб-браузерах до появления в JavaScript системы модулей, поэтому они обладают уникальной системой для включения дополнительного кода. Объект WorkerGlobalScope определяет importScripts() как глобальную функцию, к которой имеют доступ все воркеры:

```
// Перед началом работы загрузить необходимые классы и утилиты.
importScripts("utils/Histogram.js", "utils/BitSet.js");
```

Функция importScripts() принимает один или большее количество аргументов с URL, каждый из которых должен ссылаться на файл кода JavaScript. Относительные URL распознаются относительно URL, который был передан конструктору Worker() (не относительно содержащего документа). Функция importScripts() синхронно загружает и выполняет эти файлы друг за другом в порядке, в котором они были указаны. Если загрузка какого-то сценария вызывает ошибку сети или если выполнение генерирует ошибку любого рода, то ни один из последующих сценариев загружаться или выполняться не будет. Сценарий, загружаемый с помощью importScripts(), сам может вызывать importScripts() для загрузки файлов, от которых он зависит. Тем не менее, имейте в виду, что функция importScripts() не пытается отслеживать, какие сценарии уже были загружены, и ничего не делает, чтобы предотвратить циклические зависимости.

importScripts() — синхронная функция: она не возвращает управление до тех пор, пока все сценарии не будут загружены и выполнены. Вы можете начать использовать загруженные сценарии, как только произойдет возврат управления из importScripts(): нет необходимости в обратном вызове, обработчике событий методе then() или await. После того, как вы усвоили асинхронную природу JavaScript стороны клиента, кажется странным снова иметь дело с простым синхронным программированием. Но в этом и заключается прелесть

потоков: вы можете применять блокирующий вызов функции в воркере, не задерживая цикл обработки событий в главном потоке и не блокируя вычисления, которые параллельно выполняются в других воркерах.

Модули в воркерах

Чтобы использовать модули в воркерах, вам потребуется передать конструктору `Worker()` второй аргумент, которым должен быть объект со свойством `type`, установленным в строку `"module"`. Передача параметра `type:"module"` конструктору `Worker()` во многом похожа на применение атрибута `type="module"` в HTML-дескрипторе `<script>`: он означает, что код должен интерпретироваться как модуль, и разрешены объявления `import`.

Когда воркер загружает модуль, а не традиционный сценарий, объект `WorkerGlobalScope` не определяет функцию `importScripts()`.

Обратите внимание, что по состоянию на начало 2020 года Chrome был единственным браузером, который поддерживал настоящие модули и объявления `import` в воркерах.

15.13.4. Модель выполнения воркеров

Потоки воркеров выполняют свой код (и код всех импортированных сценариев или модулей) синхронно от начала до конца, после чего входят в асинхронную стадию, где реагируют на события и таймеры. Если воркер регистрирует обработчик сообщений `"message"`, то он никогда не завершит работу, пока есть вероятность поступления таких событий. Но если воркер не прослушивает сообщения, тогда он будет выполняться до тех пор, пока не исчезнут другие ожидающие задачи (такие как объекты `Promise` из `fetch()` и таймеры) и не будут вызваны все связанные с задачами обратные вызовы. После того, как вызваны все зарегистрированные обратные вызовы, у воркера не получится начать новую задачу, поэтому поток может безопасно завершить работу, что он сделает автоматически. Воркер может также сам явно прекратить работу, вызвав глобальную функцию `close()`. Обратите внимание, что в объекте `Worker` отсутствуют свойства или методы, которые определяли бы, выполняется ли поток воркера, и потому воркеры не должны самостоятельно закрывать себя без согласования этого со своим родительским потоком.

Ошибки в воркерах

Если в воркере возникает исключение, которое не перехватывается какой-либо конструкцией `catch`, тогда в глобальном объекте воркера инициируется событие `"error"`. Если такое событие обрабатывается и обработчик вызывает метод `preventDefault()` объекта события, то распространение ошибки заканчивается. В противном случае событие `"error"` инициируется в объекте `Worker`. Если здесь вызывается `preventDefault()`, тогда распространение за-

канчивается. Иначе на консоль инструментов разработчика выводится сообщение об ошибке и вызывается обработчик onerror (см. подраздел 15.1.7) объекта Window.

```
// Обработать неперехваченные ошибки воркера посредством
// обработчика внутри воркера.
self.onerror = function(e) {
   console.log(`Error in worker at ${e.filename}:${e.lineno}:
${e.message}`);
   e.preventDefault();
};
// Или обработать неперехваченные ошибки воркера с помощью
// обработчика снаружи воркера.
worker.onerror = function(e) {
   console.log(`Error in worker at ${e.filename}:${e.lineno}:
${e.message}`);
   e.preventDefault();
};
```

Подобно окнам воркеры могут регистрировать обработчик, который должен вызываться, когда объект Promise отклоняется и не предусмотрено функции .catch() для его обработки. Внутри воркера вы можете обнаруживать это путем определения функции self.onunhandledrejection или за счет использования метода addEventListener() с целью регистрации глобального обработчика для событий "unhandledrejection". Объект события, передаваемый такому обработчику, будет иметь свойство promise, чьим значением является отклоненный объект Promise, и свойство reason, значением которого будет то, что передавалось бы функции .catch().

15.13.5. postMessage(), MessagePort и MessageChannel

Метод postMessage() объекта Worker и глобальная функция postMesage(), определенная внутри воркера, работают путем вызова методов postMessage() пары объектов MessagePort, которые автоматически создаются вместе с воркером. В коде JavaScript стороны клиента нельзя напрямую обращаться к таким автоматически созданным объектам MessagePort, но с помощью конструктора MessageChannel() можно создать новую пару подключенных портов:

```
let channel = new MessageChannel;          // Создать новый канал.
let myPort = channel.port1;                // Он имеет два порта,
let yourPort = channel.port2;              // подключенные друг к другу.

myPort.postMessage("Can you hear me?"); // Сообщение,
                                           // отправленное одному,
yourPort.onmessage=(e)=>console.log(e.data);//будет получено другим.
```

MessageChannel представляет собой объект со свойствами port1 и port2, которые ссылаются на пару подключенных объектов MessagePort. Объект MessagePort содержит метод postMessage() и свойство обработчика событий onmessage. Когда метод postMessage() вызывается на одном порту подключенной пары, на другом порту пары инициируется событие "message".

Вы можете получать такие события "message", устанавливая свойство onmessage или применяя addEventListener() для регистрации прослушивателя событий "message".

Сообщения, отправленные в порт, помещаются в очередь до тех пор, пока не будет определено свойство onmessage или для порта не будет вызван метод start(), что предотвращает пропуск сообщений, посланных одним концом канала, на другом его конце. Если вы используете метод addEventListener() с объектом MessagePort, то не забудьте вызвать start(), иначе вы можете никогда не увидеть доставленное сообщение.

Во всех вызовах postMessage(), которые вы видели до сих пор, передавался единственный аргумент — объект сообщения. Но метод postMessage() также принимает необязательный второй аргумент, представляющий собой массив элементов, которые должны быть переданы на другой конец канала вместо того, чтобы иметь копию, отправленную через канал. Значениями, которые можно передавать, а не копировать, являются объекты MessagePort и ArrayBuffer. (Некоторые браузеры также реализуют другие передаваемые типы, такие как ImageBitmap и OffscreenCanvas. Однако они не поддерживаются повсеместно и в книге не рассматриваются.) Если первый аргумент postMessage() включает объект MessagePort (вложенный где-нибудь внутри объекта сообщения), тогда этот объект MessagePort также должен присутствовать во втором аргументе. Если вы поступите так, то объект MessagePort станет доступным на другом конце канала и сразу же окажется нефункциональным на вашем конце канала. Предположим, что вы создали воркер и хотите иметь два канала для взаимодействия с ним: один канал для обмена обыкновенными данными и один канал для отправки высокоприоритетных сообщений. В главном потоке вы можете создать объект MessageChannel и затем вызвать метод postMessage() воркера, чтобы передать ему один из объектов MessagePort:

```
let worker = new Worker("worker.js");
let urgentChannel = new MessageChannel();
let urgentPort = urgentChannel.port1;
worker.postMessage({ command: "setUrgentPort",
                     value: urgentChannel.port2 },
                     [ urgentChannel.port2 ]);
// Теперь мы можем получать срочные сообщения от воркера:
urgentPort.addEventListener("message", handleUrgentMessage);
urgentPort.start();  // Начать получение сообщений.

// И посылать срочные сообщения:
urgentPort.postMessage("test");
```

Объекты MessageChannel также удобны, когда вы создали два воркера и хотите позволить им взаимодействовать друг с другом напрямую, не обращаясь к коду в главном потоке для передачи сообщений между ними.

Другой способ применения второго аргумента postMessage() — передача объектов ArrayBuffer между воркерами без необходимости в их копировании. Такой прием обеспечивает значительное повышение производительности для крупных объектов ArrayBuffer вроде тех, которые хранят данные

изображений. Когда объект ArrayBuffer передается по MessagePort, то этот ArrayBuffer становится непригодным для использования в первоначальном потоке и потому возможность параллельного доступа к его содержимому отсутствует. Если первый аргумент postMessage() включает ArrayBuffer или любое значение (подобное типизированному массиву), которое содержит ArrayBuffer, тогда такой буфер может присутствовать как элемент массива во втором аргументе postMessage(). Если он присутствует, то будет передаваться без копирования, а если нет, тогда ArrayBuffer вместо передачи будет копироваться. В примере 15.14 демонстрируется применение описанной методики передачи посредством объектов ArrayBuffer.

15.13.6. Обмен сообщениями между разными источниками с помощью postMessage()

Существует еще один сценарий использования метода postMessage() в коде JavaScript стороны клиента. В него вовлечены окна, а не воркеры, но между двумя сценариями есть достаточно много общего, поэтому мы опишем здесь метод postMessage() объекта Window.

Когда документ содержит элемент <iframe>, то такой элемент действует как встроенное, но независимое окно. Объект Element, представляющий <iframe>, имеет свойство contentWindow, которое является объектом Window для встроенного документа. Для сценариев, выполняющихся внутри вложенного элемента <iframe>, свойство window.parent ссылается на содержащий объект Window. Если два окна отображают документы из того же самого источника, тогда сценарии в одном окне имеют доступ к содержимому другого окна. Но когда документы имеют разные источники, то политика одинакового источника браузера не разрешает коду JavaScript в одном окне получать доступ к содержимому другого окна.

Что касается воркеров, то метод postMessage() предлагает для двух независимых потоков безопасный способ взаимодействия без совместного использования памяти. В случае окон метод postMessage() обеспечивает управляемый способ безопасного обмена сообщениями для двух независимых источников. Даже если политика одинакового источника запрещает вашему сценарию видеть содержимое другого окна, вы все равно можете вызвать метод postMessage() другого окна, что приведет к генерации в нем события "message", которое могут видеть обработчики событий в сценариях этого окна.

Тем не менее, метод postMessage() объекта Window немного отличается от метода postMessage() объекта Worker. В первом аргументе по-прежнему передается произвольное сообщение, которое будет копироваться алгоритмом структурированного клонирования. Но необязательный второй аргумент, где перечисляются объекты, которые должны передаваться, а не копироваться, становится необязательным третьим аргументом. Метод postMessage() окна принимает строку в качестве обязательного второго аргумента. Второй аргумент должен быть источником (протокол, имя хоста и необязательный порт), который указывает ожидаемого вами получателя сообщения. Если вы переда-

дите во втором аргументе строку "https://good.example.com", но окно, которому вы отправляете сообщение, на самом деле вмещает в себе содержимое из "https://malware.example.com", тогда отправленное вами сообщение не будет доставлено. Если же вы хотите послать сообщение содержимому из любого источника, то можете передать во втором аргументе групповой символ "*".

Код JavaScript, выполняющийся внутри окна или <iframe>, может получать сообщения, отправленные этому окну или фрейму, за счет определения свойства onmessage окна или вызова метода addEventListener() для событий "message". Как и в случае воркеров, когда вы получаете событие "message" для окна, свойство data объекта события представляет собой сообщение, которое было отправлено. Однако в событиях "message", доставляемых окнам, дополнительно определены свойства source и origin. Свойство source указывает объект Window, отправивший событие, и с помощью event.source.postMessage() вы можете послать ответ. Свойство origin указывает источник содержимого в исходном окне. Отправитель сообщения не может его подделать, и когда вы получаете событие "message", то обычно будете проверять, поступило ли оно из ожидаемого источника.

15.14. Пример: множество Мандельброта

Настоящая глава, посвященная JavaScript стороны клиента, завершается длинным примером, в котором демонстрируется применение воркеров и обмена сообщениями для распараллеливания задач с большим объемом вычислений. Но он написан как привлекательное реалистичное веб-приложение и также демонстрирует использование ряда других API-интерфейсов, рассмотренных в главе, включая управление хронологией, применение класса ImageData с <canvas> и использование событий клавиатуры, указателя и изменения размера. В нем также демонстрируются важные средства базового JavaScript, в том числе генераторы и современное применение объектов Promise.

Пример представляет собой программу для отображения и изучения множества Мандельброта — сложного фрактала, который включает великолепные изображения вроде показанного на рис. 15.16.

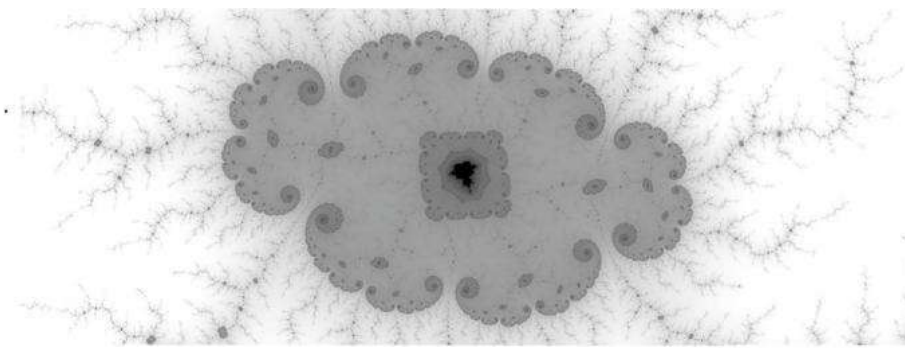


Рис. 15.16. Часть множества Мандельброта

Множество Мандельброта определяется как набор точек на комплексной плоскости, которые в результате прохождения через повторяющийся процесс комплексного умножения и сложения дают значение, чей модуль остается ограниченным. Контуры множества удивительно сложны, и установление того, какие точки являются членами множества, а какие нет, сопряжено с большим объемом вычислений: чтобы произвести изображение 500×500 множества Мандельброта, потребуется по отдельности установить членство каждого из 250 000 пикселей в изображении. А проверка того, что значение, ассоциированное с каждым пикселем, остается ограниченным, может требовать повторения процесса комплексного умножения 1000 и более раз. (Большее количество итераций обеспечивает более резко определенные границы множества; меньшее число итераций дает более размытые границы.) Имея до 250 миллионов арифметических операций с комплексными числами для синтеза высококачественного изображения множества Мандельброта, вы в состоянии понять, почему использование воркеров является очень ценной методикой. В примере 15.14 приведен код воркера, который мы будем применять. Файл относительно компактный: он содержит лишь низкоуровневые вычисления для более крупной программы. Тем не менее, в нем стоит отметить два момента.

- Воркер создает объект `ImageData`, представляющий прямоугольную сетку пикселей, для которых устанавливается членство во множестве Мандельброта. Но вместо сохранения фактических значений пикселей в `ImageData` он использует специальный типизированный массив, чтобы обходиться с каждым пикселем как с 32-битным целым числом. Он сохраняет в этот массив количество итераций, требуемых для каждого пикселя. Если модуль комплексного числа, вычисленного для каждого пикселя, становится больше четырех, тогда математически гарантировано, что с этого момента он будет неограниченно расти, и мы говорим, что он "ускользнул". Таким образом, значением, которое воркер возвращает для каждого пикселя, является количество итераций, выполняемых до того, как значение ускользнет. Мы сообщаем воркеру максимальное количество итераций, которое он должен опробовать для каждого значения, и пиксели, добирающиеся до такого максимального количества, считаются входящими во множество.
- Воркер передает объект `ArrayBuffer`, ассоциированный с `ImageData`, обратно в главный поток, так что копировать связанную с ним память не нужно.

Пример 15.14. Код воркера для расчета областей множества Мандельброта

```
// Простой воркер, который получает сообщение от своего родительского потока,
// выполняет вычисление, описанное этим сообщением, и отправляет результат
// вычисления обратно родительскому потоку.
onmessage = function(message) {
    // Сначала мы распаковываем полученное сообщение:
    //  * tile - объект со свойствами width и height. Он указывает размер
    //    прямоугольника пикселей, для которого мы будем устанавливать
```

```
//    членство во множестве Мандельброта.
//  * (x0, y0) - точка на комплексной плоскости, которая соответствует
//    левому верхнему пикселю в плитке tile.
//  * perPixel - размер пикселя в действительном и мнимом измерениях.
//  * maxIterations указывает максимальное количество итераций, которые
//    мы будем выполнять до принятия решения о членстве пикселя во множестве.
const {tile, x0, y0, perPixel, maxIterations} = message.data;
const {width, height} = tile;

//Далее мы создаем объект ImageData для представления прямоугольного массива
//пикселей,получаем его внутренний ArrayBuffer и создаем представление этого
//буфера в виде типизированного массива,чтобы можно было обходиться с каждым
// пикселем как с единым целым числом, а не с четырьмя отдельными байтами.
// В этом массиве iterations мы будем хранить количество итераций для
// каждого пикселя. (В родительском потоке итерации будут
// трансформированы в фактические цвета пикселей.)
const imageData = new ImageData(width, height);
const iterations = new Uint32Array(imageData.data.buffer);

// Теперь мы начинаем вычисление. Здесь есть три вложенных цикла.
// Два внешних цикла проходят по строкам и столбцам пикселей,
// а внутренний цикл - по каждому пикселю, чтобы выяснить,
// "ускользнул" он или нет. Ниже описаны переменные циклов:
//  * row и column - целые числа, представляющие координаты пикселя.
//  * x и y представляют комплексную точку для каждого пикселя: x + yi.
//  * index - индекс в массиве iterations для текущего пикселя.
//  * n отслеживает количество итераций для каждого пикселя.
//  * max и min отслеживают наибольшее и наименьшее количество итераций,
//    которые мы видели до сих пор для любого пикселя в прямоугольнике.
let index = 0, max = 0, min=maxIterations;
for(let row = 0, y = y0; row < height; row++, y += perPixel) {
   for(let column = 0, x = x0; column < width; column++, x += perPixel) {
      // Для каждого пикселя мы начинаем с комплексного числа c = x+yi.
      // Затем мы многократно вычисляем комплексное число z(n+1)
      // на основе следующей рекурсивной формулы:
      //    z(0) = c
      //    z(n+1) = z(n)^2 + c
      // Если |z(n)| (модуль z(n)) больше 2, тогда пиксель не является
      // частью множества и мы останавливаемся после n итераций.
      let n;              // Количество итераций, пройденных до сих пор.
      let r = x, i = y;  // Начать с z(0), установленного в c.
      for(n = 0; n < maxIterations; n++) {
         let rr = r*r, ii = i*i; // Возвести в квадрат две части z(n).
         if (rr + ii > 4) {      // Если |z(n)|^2 больше 4, тогда пиксель
            break;               // ускользнул, и мы можем остановить
                                 // итерацию.
         }
         i = 2*r*i + y;          // Вычислить мнимую часть z(n+1).
         r = rr - ii + x;        // Вычислить действительную часть z(n+1).
      }
      iterations[index++] = n;   // Запоминать количество итераций
                                 // для каждого пикселя.
      if (n > max) max = n;      // Отслеживать максимальное количество
```

```
                                        // итераций, которое мы видели,
        if (n < min) min = n;           // а также минимальное их количество.
      }
    }
    // Когда вычисление завершено, отправить результаты обратно родительскому
    // потоку. Объект imageData будет копироваться, но гигантский ArrayBuffer,
    // который в нем содержится, ради повышения производительности
    // будет передаваться.
    postMessage({tile, imageData, min, max}, [imageData.data.buffer]);
};
```

В примере 15.15 показан код приложения просмотра множества Мандельброта, которое задействует наш воркер. Теперь, когда вы добрались почти до конца этой главы, такой длинный пример будет чем-то вроде заключительного опыта, собирая вместе ряд важных средств и API-интерфейсов базового JavaScript и JavaScript стороны клиента. Код тщательно прокомментирован, и я призываю вас внимательно его прочитать.

Пример 15.15. Веб-приложение для отображения и изучения множества Мандельброта

```
/*
 * Этот класс представляет прямоугольную часть (плитку) холста
 * или изображения.
 * Мы используем объекты Tile для разделения холста на области, которые могут
 * независимо обрабатываться воркерами.
 */
class Tile {
    constructor(x, y, width, height) {
        this.x = x;                 // Свойства объекта Tile представляют
        this.y = y;                 // позицию и размер плитки внутри
        this.width = width;         // более крупного прямоугольника.
        this.height = height;
    }

    // Этот статический метод является генератором, который делит
    // прямоугольник заданной ширины и высоты на указанное количество
    // строк и столбцов и выдает numRows*numCols объектов Tile,
    // покрывающих прямоугольник.
    static *tiles(width, height, numRows, numCols) {
        let columnWidth = Math.ceil(width / numCols);
        let rowHeight = Math.ceil(height / numRows);

        for(let row = 0; row < numRows; row++) {
            let tileHeight = (row < numRows-1)
                ? rowHeight                             // высота большинства строк
                : height - rowHeight * (numRows-1);     // высота последней строки
            for(let col = 0; col < numCols; col++) {
                let tileWidth = (col < numCols-1)
                    ? columnWidth                       // ширина большинства столбцов
                    : width - columnWidth * (numCols-1); // ширина послед-
                                                         // него столбца
```

```
            yield new Tile(col * columnWidth, row * rowHeight,
                           tileWidth, tileHeight);
        }
    }
  }
}
/*
 * Этот класс представляет пул воркеров, которые все выполняют одинаковый код.
 * Указываемый вами код воркера обязан реагировать на каждое получаемое
 * сообщение,
 * выполняя какое-то вычисление и затем отправляя одиночное сообщение
 * с результатом вычисления.
 *
 * Имея объект WorkerPool и сообщение, которое представляет работу,
 * подлежащую выполнению, просто вызовите метод addWork(), передав
 * ему в аргументе это сообщение. Если есть объект Worker, который
 * в текущий момент простаивает, тогда сообщение будет немедленно
 * отправлено этому воркеру. Если простаивающих объектов Worker нет,
 * тогда сообщение ставится в очередь и будет отправлено воркеру,
 * который стал доступным.
 *
 * Метод addWork() возвращает объект Promise, который будет разрешен
 * с сообщением, полученным от работы, или отклонен, если воркер
 * сгенерировал необработанную ошибку.
 */
class WorkerPool {
  constructor(numWorkers, workerSource) {
    this.idleWorkers = [];          // Воркеры, которые в текущий момент
                                    // простаивают.
    this.workQueue = [];            // Работа, которая в текущий момент
                                    // не выполняется.
    this.workerMap = new Map();     // Отображает воркеры на функции
                                    // разрешения и отклонения.

    // Создать указанное количество воркеров, добавить обработчики
    // сообщений и ошибок и сохранить их в массиве idleWorkers.
    for(let i = 0; i < numWorkers; i++) {
      let worker = new Worker(workerSource);
      worker.onmessage = message => {
        this._workerDone(worker, null, message.data);
      };
      worker.onerror = error => {
        this._workerDone(worker, error, null);
      };
      this.idleWorkers[i] = worker;
    }
  }

  // Этот внутренний метод вызывается, когда воркер завершает работу,
  // либо посылая сообщение, либо генерируя ошибку.
  _workerDone(worker, error, response) {
    // Найти функции resolve() и reject() для этого
    // воркера и удалить запись воркера из workerMap.
```

```
        let [resolver, rejector] = this.workerMap.get(worker);
        this.workerMap.delete(worker);

        // Если нет ожидающей в очереди работы, тогда поместить этот
        // воркер в список простаивающих воркеров. В противном случае
        // извлечь работу из очереди и оправить этому воркеру.
        if (this.workQueue.length === 0) {
            this.idleWorkers.push(worker);
        } else {
            let [work, resolver, rejector] = this.workQueue.shift();
            this.workerMap.set(worker, [resolver, rejector]);
            worker.postMessage(work);
        }

        // В заключение разрешить или отклонить объект Promise,
        // ассоциированный с воркером.
        error === null ? resolver(response) : rejector(error);
    }

    // Этот метод добавляет работу в пул воркеров и возвращает объект Promise,
    // который будет разрешен с ответом воркера, когда работа сделана. Работа
    // представляет собой значение, которое нужно передать воркеру с помощью
    // postMessage(). При наличии простаивающего воркера сообщение с работой
    // будет послано незамедлительно, а иначе работа будет находиться
    // в очереди, пока воркер не станет доступным.
    addWork(work) {
        return new Promise((resolve, reject) => {
            if (this.idleWorkers.length > 0) {
                let worker = this.idleWorkers.pop();
                this.workerMap.set(worker, [resolve, reject]);
                worker.postMessage(work);
            } else {
                this.workQueue.push([work, resolve, reject]);
            }
        });
    }
}

/*
 * Этот класс хранит информацию состояния, необходимую для визуализации
 * множества Мандельброта. Свойства cx и cy дают точку на комплексной
 * плоскости, которая является центром изображения. Свойство perPixel
 * указывает, насколько действительная и мнимая части комплексного числа
 * изменяются для каждого пикселя изображения. Свойство maxIterations
 * указывает, насколько объемной должна быть работа для вычисления множества.
 * Более высокие числа требуют большего объема вычислений, но производят
 * более четкие изображения. Обратите внимание, что размер холста не входит
 * в состояние. Для заданных cx, cy и perPixel мы просто визуализируем любую
 * порцию множества Мандельброта, умещающуюся в холст при его текущем размере.
 *
 * Объекты этого типа используются с методом history.pushState() и применяются
 * для чтения желаемого состояния из помещенного в закладки или разделяемого URL
 */
class PageState {
```

```
    // Этот фабричный метод возвращает начальное состояние
    // для отображения полного множества.
    static initialState() {
        let s = new PageState();
        s.cx = -0.5;
        s.cy = 0;
        s.perPixel = 3/window.innerHeight;
        s.maxIterations = 500;
        return s;
    }

    // Этот фабричный метод добывает состояние из URL или возвращает null,
    // если из URL не удается прочитать допустимое состояние.
    static fromURL(url) {
        let s = new PageState();
        let u = new URL(url);      // Инициализировать состояние
                                   // из параметров поиска URL.
        s.cx = parseFloat(u.searchParams.get("cx"));
        s.cy = parseFloat(u.searchParams.get("cy"));
        s.perPixel = parseFloat(u.searchParams.get("pp"));
        s.maxIterations = parseInt(u.searchParams.get("it"));
        // Если мы получили допустимые значения, тогда возвратить
        // объект PageState, а иначе null.
        return (isNaN(s.cx) || isNaN(s.cy) || isNaN(s.perPixel)
                || isNaN(s.maxIterations))
            ? null
            : s;
    }

    // Этот метод экземпляра кодирует текущее состояние в виде
    // параметров поиска для текущего местоположения браузера.
    toURL() {
        let u = new URL(window.location);
        u.searchParams.set("cx", this.cx);
        u.searchParams.set("cy", this.cy);
        u.searchParams.set("pp", this.perPixel);
        u.searchParams.set("it", this.maxIterations);
        return u.href;
    }
}

// Эти константы управляют параллелизмом вычислений множества Мандельброта.
// Вам может потребоваться подрегулировать их, чтобы получить оптимальную
// производительность на своем компьютере.
const ROWS = 3, COLS = 4, NUMWORKERS = navigator.hardwareConcurrency || 2;

// Это главный класс нашей программы для отображения множества Мандельброта.
// Просто вызовите функцию конструктора с элементом <canvas>, куда должна
// выполняться визуализация. В программе предполагается, что целевой элемент
// <canvas> стилизован так, что имеет размеры окна браузера.
class MandelbrotCanvas {
    constructor(canvas) {
        // Сохранить холст, получить его объект контекста
        // и инициализировать WorkerPool.
```

```
    this.canvas = canvas;
    this.context = canvas.getContext("2d");
    this.workerPool = new WorkerPool(NUMWORKERS, "mandelbrotWorker.js");

    // Определить ряд свойств для будущего использования.
    this.tiles = null;              // Подобласти холста.
    this.pendingRender = null;      // В текущий момент мы не визуализируем.
    this.wantsRerender = false;     // Визуализация в текущий момент
                                    // не запрашивалась.
    this.resizeTimer = null;        // Препятствует слишком частому
                                    // изменению размера.
    this.colorTable = null;         // Для преобразования низкоуровневых
                                    // данных в значения пикселей.

    // Настроить наши обработчики событий.
    this.canvas.addEventListener("pointerdown",
                                 e => this.handlePointer(e));
    window.addEventListener("keydown", e => this.handleKey(e));
    window.addEventListener("resize", e => this.handleResize(e));
    window.addEventListener("popstate", e =>this.setState(e.state, false));

    // Инициализировать наше состояние из URL или воспользоваться
    // начальным состоянием.
    this.state =
       PageState.fromURL(window.location) || PageState.initialState();

    // Сохранить это состояние с помощью механизма хронологии.
    history.replaceState(this.state, "", this.state.toURL());

    // Установить размер холста и получить массив плиток,
    // которые его покрывают.
    this.setSize();

    // И визуализировать множество Мандельброта внутри холста.
    this.render();
}

// Установить размер холста и инициализировать массив объектов Tile.
// Этот метод вызывается из конструктора и также методом handleResize(),
// когда изменяются размеры окна браузера.
setSize() {
    this.width = this.canvas.width = window.innerWidth;
    this.height = this.canvas.height = window.innerHeight;
    this.tiles = [...Tile.tiles(this.width, this.height, ROWS, COLS)];
}

// Эта функция вносит изменение в PageState, повторно визуализирует
// множество Мандельброта, используя новое состояние, и сохраняет
// новое состояние посредством history.pushState(). Если в первом
// аргументе передается функция, то эта функция будет вызываться
// с объектом состояния в качестве ее аргумента и должна вносить
// изменения в состояние. Если в первом аргументе передается объект,
// то мы просто копируем свойства этого объекта в объект состояния.
// Если необязательный второй аргумент равен false, тогда новое
// состояние не будет сохраняться. (Мы делаем это при вызове
// setState() в ответ на событие "popstate".)
setState(f, save=true) {
```

```
    // Если аргумент - функция, тогда вызвать ее для обновления состояния.
    // В противном случае копировать его свойства в текущее состояние.
    if (typeof f === "function") {
        f(this.state);
    } else {
        for(let property in f) {
            this.state[property] = f[property];
        }
    }

    // В любом случае как можно скорее начать визуализацию
    // нового состояния.
    this.render();

    // Обычно мы сохраняем новое состояние за исключением ситуации,
    // когда второй аргумент равен false, что делается в случае
    // получения события "popstate".
    if (save) {
        history.pushState(this.state, "", this.state.toURL());
    }
}

// Этот метод асинхронно вычерчивает на холсте часть множества
// Мандельброта, указанную объектом PageState. Он вызывается
// конструктором, методом setState(), когда изменяется состояние,
// и обработчиком события "resize", когда изменяется размер холста.
render() {
    // Иногда пользователь может использовать клавиатуру или мышь
    // для запрашивания визуализации быстрее, чем мы можем ее выполнить.
    // Мы не хотим отправлять все запросы визуализации в пул воркеров.
    // Взамен, если мы выполняем визуализацию, то просто помечаем,
    // что необходима новая визуализация, и когда текущая визуализация
    // завершается, мы визуализируем текущее состояние, возможно
    // пропустив множество промежуточных состояний.
    if (this.pendingRender) {      // Если мы уже выполняем визуализацию,
        this.wantsRerender = true; // тогда отметить необходимость
                                   // визуализировать позже,
        return;                    // а сейчас больше ничего не делать.
    }

    // Получить наши переменные состояния и рассчитать
    // комплексное число для левого верхнего угла холста.
    let {cx, cy, perPixel, maxIterations} = this.state;
    let x0 = cx - perPixel * this.width/2;
    let y0 = cy - perPixel * this.height/2;

    // Для каждой из ROWS*COLS плиток вызывать addWork() с сообщением
    // для кода в mandelbrotWorker.js. Накапливать результирующие
    // объекты Promise в массиве.
    let promises = this.tiles.map(tile => this.workerPool.addWork({
        tile: tile,
        x0: x0 + tile.x * perPixel,
        y0: y0 + tile.y * perPixel,
        perPixel: perPixel,
        maxIterations: maxIterations
    }));
```

```
// Использовать Promise.all() для получения массива ответов
// из массива объектов Promise. Каждый ответ является вычислением
// для одной из наших плиток. Вспомните из кода mandelbrotWorker.js,
// что каждый ответ включает объект Tile, объект ImageData,
// содержащий счетчики итераций вместо значений пикселей, а также
// минимальное и максимальное количество итераций для этой плитки.
this.pendingRender = Promise.all(promises).then(responses => {
    // Первым делом найти общие минимальное и максимальное количества
    // итераций по всем плиткам. Эти числа нам необходимы, чтобы можно
    // было назначить цвета пикселям.
    let min = maxIterations, max = 0;
    for(let r of responses) {
        if (r.min < min) min = r.min;
        if (r.max > max) max = r.max;
    }

    // Теперь нам нужен способ преобразования низкоуровневых счетчиков
    // итераций из воркеров в цвета пикселей, которые будут отображаться
    // на холсте. Нам известно, что все пиксели имеют количество
    // итераций между минимальным и максимальным, поэтому мы
    // предварительно вычисляем цвета для каждого счетчика итераций
    // и сохраняем их в массиве colorTable.

    // Если мы пока еще не создали таблицу цветов или она больше
    // не имеет правильный размер, тогда сделать это сейчас.
    if (!this.colorTable ||
         this.colorTable.length !== maxIterations+1){
        this.colorTable = new Uint32Array(maxIterations+1);
    }

    // Имея максимальное и минимальное количество итераций,
    // рассчитать соответствующие значения в таблице цветов.
    // Пиксели внутри множества будут окрашены в полностью
    // непрозрачный черный цвет. Пиксели вне множества будут иметь
    // полупрозрачный черный цвет, причем с более высокими
    // значениями счетчиков итераций связана большая непрозрачность.
    // Пиксели с минимальными счетчиками итераций будут прозрачными,
    // и через них будет виден белый фон, что в итоге дает изображение
    // в оттенках серого.
    if (min === max) {                  // Если все пиксели одинаковы,
        if (min === maxIterations) {// тогда сделать их всех черными
            this.colorTable[min] = 0xFF000000;
        } else {                        // или прозрачными.
            this.colorTable[min] = 0;
        }
    } else {
        // В нормальном случае, когда min и max разные, использовать
        // логарифмическую шкалу, чтобы назначить каждому возможному
        // счетчику итераций непрозрачность между 0 и 255, после чего
        // применить операцию сдвига влево для превращения
        // непрозрачности в значение пикселя.
        let maxlog = Math.log(1+max-min);
        for(let i = min; i <= max; i++) {
            this.colorTable[i] =
```

```
                (Math.ceil(Math.log(1+i-min)/maxlog * 255) << 24);
            }
        }

        // Транслировать количества итераций в ImageData
        // каждого ответа в цвета из colorTable.
        for(let r of responses) {
            let iterations = new Uint32Array(r.imageData.data.buffer);
            for(let i = 0; i < iterations.length; i++) {
                iterations[i] = this.colorTable[iterations[i]];
            }
        }

        // Наконец, визуализировать все объекты imageData
        // в соответствующие им плитки холста, используя putImageData().
        // (Однако сначала удалить любые трансформации CSS холста, которые
        // могли быть установлены обработчиком событий "pointerdown".)
        this.canvas.style.transform = "";
        for(let r of responses) {
            this.context.putImageData(r.imageData, r.tile.x, r.tile.y);
        }
    })
    .catch((reason) => {
        // Если что-то пошло не так в любом объекте Promise, тогда мы
        // выводим здесь сообщение об ошибке. Это не должно происходить,
        // но поможет в отладке, если все-таки случится.
        console.error("Promise rejected in render():", reason);
                    // Promise отклонен в render()
    })
    .finally(() => {
        // Когда мы закончили визуализацию, очистить флаги pendingRender.
        this.pendingRender = null;

        // И если поступали запросы на визуализацию, пока мы были заняты,
        // тогда выполнить визуализацию прямо сейчас.
        if (this.wantsRerender) {
            this.wantsRerender = false;
            this.render();
        }
    });
}

// Если пользователь изменяет размеры окна, то эта функция будет
// вызываться многократно. Изменение размера холста и повторная
// визуализация множества Мандельброта является затратной операцией,
// которую мы не можем делать несколько раз в секунду, поэтому мы
// используем таймер для задержки обработки изменения размеров, пока
// не истекут 200 мс с момента получения последнего события "resize".
handleResize(event) {
    // Если мы уже откладывали запрос на изменение размеров,
    // тогда очистить его.
    if (this.resizeTimer) clearTimeout(this.resizeTimer);

    // И взамен отложить этот запрос изменение размеров.
    this.resizeTimer = setTimeout(() => {
```

```
            this.resizeTimer = null;   // Пометить, что изменение
                                       // размеров обработано.
            this.setSize();  // Изменить размеры холста и плиток.
            this.render();   // Повторно визуализировать с новыми размерами.
        }, 200);
    }
    // Если пользователь нажимает клавишу, будет вызван этот обработчик событий.
    // Мы вызываем setState() в ответ на нажатие различных клавиш,
    // и setState() визуализирует новое состояние, обновляет URL
    // и сохраняет состояние в хронологии браузера.
    handleKey(event) {
        switch(event.key) {
        case "Escape":  //Нажмите <Esc>,чтобы вернуться в начальное состояние.
            this.setState(PageState.initialState());
            break;
        case "+":       // Нажмите <+>, чтобы увеличить количество итераций.
            this.setState(s => {
                s.maxIterations = Math.round(s.maxIterations*1.5);
            });
            break;
        case "-":       // Нажмите <->, чтобы уменьшить количество итераций.
            this.setState(s => {
                s.maxIterations = Math.round(s.maxIterations/1.5);
                if (s.maxIterations < 1) s.maxIterations = 1;
            });
            break;
        case "o":           // Нажмите <o>, чтобы уменьшить масштаб.
            this.setState(s => s.perPixel *= 2);
            break;
        case "ArrowUp":     // Нажмите клавишу со стрелкой вверх
                            // для прокрутки вверх.
            this.setState(s => s.cy -= this.height/10 * s.perPixel);
            break;
        case "ArrowDown": // Нажмите клавишу со стрелкой вниз для прокрутки вниз.
            this.setState(s => s.cy += this.height/10 * s.perPixel);
            break;
        case "ArrowLeft":   // Нажмите клавишу со стрелкой влево
                            // для прокрутки влево.
            this.setState(s => s.cx -= this.width/10 * s.perPixel);
            break;
        case "ArrowRight":  // Нажмите клавишу со стрелкой вправо
                            // для прокрутки вправо.
            this.setState(s => s.cx += this.width/10 * s.perPixel);
            break;
        }
    }
    // Этот метод вызывается,когда мы получаем событие "pointerdown" для холста.
    // Событие "pointerdown" может быть началом жеста масштабирования (щелчка
    // или касания) или жеста прокрутки (перетаскивания). Данный обработчик
    // регистрирует обработчики для событий "pointermove" и "pointerup", чтобы
    // реагировать на остаток жеста. (Эти два добавочных обработчика удаляются,
    // когда жест заканчивается событием "pointerup".)
```

```
handlePointer(event) {
    // Пиксельные координаты и время начального события "pointerdown".
    // Поскольку холст имеет размеры окна, координаты в событии также
    // будут координатами на холсте.
    const x0 = event.clientX, y0 = event.clientY, t0 = Date.now();

    // Обработчик для событий "pointermove".
    const pointerMoveHandler = event => {
        // Насколько далеко переместился указатель
        // и сколько времени прошло?
        let dx=event.clientX-x0, dy=event.clientY-y0, dt=Date.now()-t0;

        // Если указатель переместился достаточно далеко или прошло
        // достаточно времени, тогда это не обыкновенный щелчок,
        // поэтому мы применяем CSS для перетаскивания. (Мы его повторно
        // визуализируем, когда получим событие "pointerup".)
        if (dx > 10 || dy > 10 || dt > 500) {
            this.canvas.style.transform = `translate(${dx}px, ${dy}px)`;
        }
    };

    // Обработчик для событий "pointerup".
    const pointerUpHandler = event => {
        // Когда указатель поднимается вверх, жест закончен, поэтому удалить
        // обработчики для "pointermove" и "pointerup" до следующего жеста.
        this.canvas.removeEventListener("pointermove", pointerMoveHandler);
        this.canvas.removeEventListener("pointerup", pointerUpHandler);

        // Насколько далеко переместился указатель
        // и сколько времени прошло?
        const dx = event.clientX-x0, dy=event.clientY-y0, dt=Date.now()-t0;
        // Распаковать объект состояния в индивидуальные константы.
        const {cx, cy, perPixel} = this.state;

        // Если указатель переместился достаточно далеко или прошло
        // достаточно времени, тогда это был жест перетаскивания,
        // и нам необходимо модифицировать состояние, чтобы изменить
        // точку центра. В противном случае пользователь щелкнул или
        // коснулся точки и нам нужно центрировать относительно этой точки
        // и увеличить масштаб.
        if (dx > 10 || dy > 10 || dt > 500) {
            // Пользователь прокрутил изображение на (dx, dy) пикселей.
            // Преобразовать эти значения в смещения на комплексной
            // плоскости.
            this.setState({cx: cx - dx*perPixel, cy: cy - dy*perPixel});
        } else {
            // Пользователь выполнил щелчок. Подсчитать,
            // на сколько пикселей переместился центр.
            let cdx = x0 - this.width/2;
            let cdy = y0 - this.height/2;

            // Использовать CSS для быстрого и временного увеличения
            // масштаба.
            this.canvas.style.transform =
                `translate(${-cdx*2}px, ${-cdy*2}px) scale(2)`;
```

```
            // Установить комплексные координаты новой точки центра
            // и увеличить масштаб в 2 раза.
            this.setState(s => {
               s.cx += cdx * s.perPixel;
               s.cy += cdy * s.perPixel;
               s.perPixel /= 2;
            });
         }
      };
      // Когда пользователь начинает жест, мы регистрируем обработчики
      // для событий "pointermove" и "pointerup", которые будут следующими.
      this.canvas.addEventListener("pointermove", pointerMoveHandler);
      this.canvas.addEventListener("pointerup", pointerUpHandler);
   }
}

// Наконец, вот как мы настраиваем холст. Обратите внимание, что данный файл
// JavaScript самодостаточен. Файл HTML необходим только для включения этого
// одного дескриптора <script>.
let canvas = document.createElement("canvas"); // Создать элемент холста.
document.body.append(canvas);                  // Вставить его в тело.
document.body.style = "margin:0";              // Поля для <body> отсутствуют.
canvas.style.width = "100%";                   // Сделать холст настолько же
canvas.style.height = "100%";                  // широким и высоким как тело.
new MandelbrotCanvas(canvas);                  // И начать в нем визуализацию!
```

15.15. Резюме и рекомендации относительно дальнейшего чтения

В этой длинной главе были раскрыты основы программирования на JavaScript стороны клиента.

- Способы включения сценариев и модулей JavaScript в веб-страницы, а также как и когда они выполняются.
- Асинхронная и управляемая событиями программная модель JavaScript стороны клиента.
- Объектная модель документа (Document Object Model — DOM), которая позволяет коду JavaScript инспектировать и модифицировать HTML-содержимое документа, внутрь которого оно встроено. API-интерфейс DOM является центральной частью программирования на JavaScript стороны клиента.
- Возможность манипулирования в коде JavaScript стилями CSS, которые применены к содержимому внутри документа.
- Возможность получения в коде JavaScript координат элементов документа в окне браузера и внутри самого документа.

- Способы создания многократно используемых "веб-компонентов" пользовательского интерфейса с помощью JavaScript, HTML и CSS, применяя специальные элементы и API-интерфейсов теневой модели DOM.
- Способы отображения и динамической генерации графики посредством SVG и HTML-элемента <canvas>.
- Способы добавления в коде звуковых эффектов (записанных и синтезированных) к веб-страницам.
- Как JavaScript может заставить браузер загружать новые страницы, перемещаться назад и вперед в хронологии просмотра и даже добавлять в хронологию просмотра новые записи.
- Как программы JavaScript могут обмениваться данными с веб-серверами, используя протоколы HTTP и WebSocket.
- Как программы JavaScript могут сохранять данные в браузере пользователя.
- Как программы JavaScript могут задействовать потоки воркеров для обеспечения безопасной формы параллелизма.

Безусловно, это была самая длинная глава в книге. Но она далека от того, чтобы охватить все API-интерфейсы, доступные веб-браузерам. Веб-платформа разрастается и постоянно эволюционирует, и моей целью в главе было представление наиболее важных основных API-интерфейсов. Благодаря знаниям, которые у вас появятся после чтения настоящей книги, вы будете хорошо подготовлены к изучению и использованию новых API-интерфейсов, когда возникнет такая необходимость. Но вы не сможете изучить новый API-интерфейс, не зная о его существовании, так что в последующих подразделах приведен краткий список средств веб-платформы, которые вам, возможно, придется исследовать в будущем.

15.15.1. HTML и CSS

Веб-сеть построена на основе трех ключевых технологий: HTML, CSS и JavaScript, и знание JavaScript может сделать вас лишь разработчиком веб-приложений, если только вы не разовьете свой опыт работы с HTML и CSS. Важно знать, как применять JavaScript для манипулирования HTML-элементами и CSS-стилями, но эти знания гораздо более полезны, если вы также осведомлены о том, какие HTML-элементы и CSS-стили должны использоваться.

Таким образом, прежде чем приступать к исследованию дополнительных API-интерфейсов для JavaScript, я советую вам потратить какое-то время на овладение другими инструментами в инструментальном комплекте разработчика веб-приложений. Скажем, HTML-форма и элементы ввода имеют сложное поведение, которое важно понимать, а режимы компоновки типа флексбокса и сетки в CSS обладают невероятной мощью.

В этой области особого внимания заслуживают две темы — доступность (включая атрибуты ARIA) и интернационализация (в том числе поддержка написания справа налево).

15.15.2. Производительность

После того, как вы написали веб-приложение и представили его миру, начнется нескончаемый поиск способа сделать его быстрее. Однако сложно оптимизировать то, что невозможно измерить, поэтому вам стоит ознакомиться с API-интерфейсом Performance. Главной точкой входа в данном API-интерфейсе является свойство `performance` объекта окна. Он включает источник времени с высоким разрешением `performance.now()`, а также методы `performance.mark()` и `performance.measure()` для пометки критических точек в вашем коде и измерения прошедшего между ними времени. Вызов указанных методов создает объекты `PerformanceEntry`, доступ к которым вы можете получить доступ посредством метода `performance.getEntries()`.

Браузеры добавляют собственные объекты `PerformanceEntry` каждый раз, когда браузер загружает новую страницу либо извлекает файл через сеть, и такие автоматически создаваемые объекты `PerformanceEntry` включают подробные детали хронометража сетевой производительности вашего приложения. Связанный класс `PerformanceObserver` позволяет указывать функцию, которая должна вызываться, когда создаются новые объекты `PerformanceEntry`.

15.15.3. Безопасность

В главе было дано общее представление о том, как защищать веб-сайты от уязвимостей типа межсайтовых сценариев (XSS), но мы не вдавались в особые детали. Тема веб-безопасности важна и вы можете потратить некоторое время на ее изучение. Помимо XSS полезно знать о HTTP-заголовке `Content-Security-Policy` и понимать, как он позволяет запрашивать у веб-браузера ограничение возможностей, которые он предоставляет коду JavaScript. В той же степени важно понимать CORS.

15.15.4. WebAssembly

WebAssembly (или "wasm") — это низкоуровневый формат байт-кода виртуальной машины, который предназначен для эффективной интеграции с интерпретаторами JavaScript в веб-браузерах. Существуют компиляторы, которые позволяют компилировать программы на C, C++ и Rust в байт-код WebAssembly и выполнять результирующие программы в веб-браузерах со скоростью, близкой к присущей им изначально, не разрушая песочницу браузера или модель безопасности.

WebAssembly может экспортировать функции, которые разрешено вызывать в программах JavaScript. Типичным вариантом применения WebAssembly была бы компиляция стандартной библиотеки сжатия zlib на языке C, чтобы код JavaScript имел доступ к высокоскоростным алгоритмам сжатия и распаковки. Дополнительные сведения ищите на веб-сайте `https://webassembly.org/`.

15.15.5. Дополнительные средства объектов Document и Window

Объекты Window и Document имеют несколько функциональных средств, которые в главе не рассматривались.

- В объекте Window определены методы alert(), confirm() и prompt(), которые отображают для пользователя простые модальные диалоговые окна. Указанные методы блокируют главный поток. Метод confirm() синхронно возвращает булевское значение, а prompt() синхронно возвращает строку пользовательского ввода. Они не подходят для производственной среды, но удобны для создания простых проектов и прототипов.
- Свойства navigator и screen объекта Window вскользь упоминались в начале главы, но объекты Navigator и Screen, на которые они ссылаются, обладают рядом не раскрытых здесь характеристик, представляющих определенный интерес.
- Метод requestFullscreen() любого объекта Element запрашивает отображение этого элемента (<video> или <canvas>, например) в полноэкранном режиме. Метод exitFullscreen() объекта Document обеспечивает возврат к нормальному режиму отображения.
- Метод requestAnimationFrame() объекта Window принимает функцию в качестве своего аргумента и будет выполнять ее, когда браузер готовится визуализировать следующий кадр. В случае внесения визуальных изменений (особенно повторяющихся или анимационных) помещение вашего кода внутрь вызова requestAnimationFrame() может помочь в обеспечении того, что изменения визуализируются плавно и оптимизированы браузером.
- Если пользователь выбирает текст внутри вашего документа, то вы можете выяснить детали выбора посредством метода getSelection() объекта Window и получить выбранный текст с помощью getSelection().toString(). В некоторых браузерах navigator.clipboard является объектом с асинхронным API-интерфейсом для чтения и установки содержимого буфера обмена системы, чтобы сделать возможными взаимодействия с приложениями вне браузера.
- Малоизвестная особенность веб-браузеров связана с тем, что HTML-элементы с атрибутом contenteditable="true" позволяют редактировать свое содержимое. Метод document.execCommand() включает средства расширенного редактирования текста для редактируемого содержимого.
- Объект MutationObserver позволяет коду JavaScript следить за изменениями в указанном элементе документа или ниже него. Создайте объект MutationObserver посредством конструктора MutationObserver(), передав функцию обратного вызова, которая должна вызываться, когда делаются изменения. Затем вызовите метод observe() объекта MutationObserver, чтобы указать, за какими частями каких элементов нужно следить.

- Объект `IntersectionObserver` позволяет коду JavaScript выяснять, какие элементы документа находятся на экране, а какие близки к тому, чтобы находиться на экране. Это особенно полезно для приложений, которые хотят динамически загружать содержимое по запросу, когда пользователь выполняет прокрутку.

15.15.6. События

Количество и разнообразие событий, поддерживаемых веб-платформой, может приводить в растерянность. В главе обсуждались различные типы событий, но ниже перечислено еще несколько, которые вы можете счесть полезными.

- Браузеры инициируют события `"online"` и `"offline"` в объекте `Window`, когда браузер получает или утрачивает подключение к Интернету.
- Браузеры инициируют событие `"visiblitychange"` в объекте `Document`, когда документ становится видимым или невидимым (обычно из-за того, что пользователь переходит с вкладки на вкладку). В коде JavaScript можно проверять `document.visibilityState` для определения, виден ли (`"visible"`) документ в текущий момент или же он скрыт (`"hidden"`).
- Браузеры поддерживают замысловатый API-интерфейс для пользовательских интерфейсов с перетаскиванием и обмена данными с приложениями за пределами браузера. В этом API-интерфейсе задействовано несколько событий, в частности `"dragstart"`, `"dragover"`, `"dragend"` и `"drop"`. Данный API-интерфейс сложно использовать корректно, но он полезен, когда нужен. О нем важно знать, если вы хотите предоставить пользователям возможность перетаскивания файлов со своего рабочего стола в ваше веб-приложение.
- API-интерфейс Pointer Lock позволяет коду JavaScript скрывать указатель мыши и получать низкоуровневые события мыши в виде относительных величин перемещения, а не абсолютных позиций на экране. Обычно поступать так полезно в играх. Вызовите метод `requestPointerLock()` для элемента, которому желаете направлять все события мыши. Затем события `"mousemove"`, доставленные этому элементу, будут иметь свойства `movementX` и `movementY`.
- API-интерфейс Gamepad добавляет поддержку для игровых контроллеров. Применяйте `navigator.getGamepads()` для получения подключенных объектов `Gamepad` и прослушивайте события `"gamepadconnected"` в объекте `Window`, чтобы получать уведомления при подключении нового контроллера. Объект `Gamepad` определяет API-интерфейс для запрашивания текущего состояния кнопок в контроллере.

15.15.7. Прогрессивные веб-приложения и служебные воркеры

Термин *прогрессивные веб-приложения* (Progressive Web App — PWA) представляет собой специальный термин, который описывает веб-приложения, построенные с использованием ряда ключевых технологий. Тщательная документация таких ключевых технологий потребовала бы целой книги, и в настоящей главе они не рассматриваются, но вы должны быть осведомлены обо всех этих API-интерфейсах. Стоит отметить, что мощные современные API-интерфейсы подобного рода обычно спроектированы для работы только с защищенными подключениями HTTPS. Веб-сайты, которые по-прежнему применяют URL вида `http://`, не смогут воспользоваться следующими преимуществами.

- `ServiceWorker` является потоком воркера, способным перехватывать, инспектировать и отвечать на сетевые запросы из веб-приложения, которое он "обслуживает". Когда веб-приложение регистрирует служебный воркер, код такого воркера становится постоянным в локальном хранилище браузера и при повторном посещении пользователем веб-сайта служебный воркер снова активизируется. Служебные воркеры могут кешировать сетевые ответы (включая файл кода JavaScript), а это значит, что веб-приложения, которые задействуют служебные воркеры, способны эффективно устанавливать сами себя на компьютере пользователя для быстрого запуска и автономного применения. На веб-сайте `https://serviceworke.rs` предлагается книга *Service Worker Cookbook* — ценный ресурс, который дает возможность изучить служебные воркеры и связанные с ними технологии.
- API-интерфейс Cache предназначен для использования со служебными воркерами (но также доступен обыкновенному коду JavaScript за рамками воркеров). Он работает с объектами `Request` и `Response`, определенными API-интерфейсом `fetch()`, и реализует кеш пар `Request/Response`. API-интерфейс Cache позволяет служебному воркеру кешировать сценарии и другие ресурсы веб-приложения, которое он обслуживает, и также может содействовать автономному применению веб-приложения (что особенно важно для мобильных устройств).
- Файл веб-манифеста (Web Manifest) имеет формат JSON и описывает веб-приложение, включая название, URL и ссылки на значки различных размеров. Если ваше веб-приложение использует служебный воркер и содержит дескриптор `<link rel="manifest">`, который ссылается на файл `.webmanifest`, тогда браузеры (особенно браузеры на мобильных устройствах) могут предоставить вам возможность добавить значок для веб-приложения на рабочий стол или домашний экран.
- API-интерфейс Notifications позволяет веб-приложениям отображать уведомления с применением собственной системы уведомлений операционной системы на мобильных и настольных устройствах. Уведомления мо-

гут включать изображение и текст, а ваш код может получать событие, когда пользователь щелкает на уведомлении. Использование этого API-интерфейса осложняется тем фактом, что вы обязаны сначала запросить у пользователя разрешение на отображение уведомлений.

- API-интерфейс Push позволяет веб-приложениям, которые имеют служебный воркер (и разрешение от пользователя), подписываться на уведомления от сервера и отображать их, даже когда само веб-приложение не запущено. Push-уведомления распространены на мобильных устройствах, и API-интерфейс Push приближает функциональность веб-приложений к функциональности собственных приложений мобильного устройства.

15.15.8. API-интерфейсы мобильных устройств

Существует несколько API-интерфейсов, которые в первую очередь полезны для веб-приложений, выполняющихся на мобильных устройствах. (К сожалению, ряд таких API-интерфейсов работает только на устройствах Android, но не устройствах iOS.)

- API-интерфейс Geolocation позволяет коду JavaScript (при наличии разрешения от пользователя) определять физическое местоположение пользователя. Он хорошо поддерживается на настольных и мобильных устройствах, в том числе на устройствах iOS. Применяйте `navigator.geolocation.getCurrentPosition()` для запроса текущего местоположения пользователя и `navigator.geolocation.watchPosition()` для регистрации обратного вызова, который должен вызываться, когда местоположение пользователя изменяется.
- Метод `navigator.vibrate()` заставляет мобильное устройство (но не iOS) вибрировать. Зачастую это разрешено только в ответ на пользовательский жест, но вызов метода `navigator.vibrate()` позволит вашему приложению беззвучно сообщать о том, что жест был распознан.
- API-интерфейс ScreenOrientation предоставляет веб-приложению возможность запрашивать текущую ориентацию экрана мобильного устройства и также блокировать себя в альбомной или портретной ориентации.
- События `"devicemotion"` и `"deviceorientation"` в объекте окна сообщают данные акселерометра и магнитометра для устройства, позволяя вам определять, насколько устройство ускоряется и как пользователь ориентирует его в пространстве. (События работают под управлением iOS.)
- API-интерфейс Sensor пока еще широко не поддерживается помимо Chrome на устройствах Android, но он предоставляет коду JavaScript возможность доступа к полному комплекту датчиков мобильного устройства, включая акселерометр, магнитометр и датчик окружающего света. Указанные датчики позволяют коду JavaScript определять, например, в каком направлении смотрит пользователь, или обнаруживать, когда пользователь встряхивает свой телефон.

15.15.9. API-интерфейсы для работы с двоичными данными

Типизированные массивы, объекты `ArrayBuffer` и класс `DataView` (все они раскрывались в разделе 11.2) позволяют коду JavaScript работать с двоичными данными. Как объяснялось ранее в главе, API-интерфейс `fetch()` дает возможность программам JavaScript загружать двоичные данные по сети. Еще одним источником двоичных данных являются файлы из локальной файловой системы пользователя. По причинам, связанным с безопасностью, код JavaScript не может просто читать локальные файлы. Но если пользователь выбирает файл для выгрузки (используя элемент формы `<input type="file">`) или применяет перетаскивание для помещения файла внутрь вашего веб-приложения, тогда код JavaScript может получать доступ к файлу как к объекту `File`.

`File` — это подкласс класса `Blob`, и как таковой, он будет непрозрачным представлением порции данных. Вы можете использовать класс `FileReader` для асинхронного получения содержимого файла в виде объекта `ArrayBuffer` или строки. (В некоторых браузерах вы можете пропустить `FileReader` и взамен применять основанные на `Promise` методы `text()` и `arrayBuffer()`, определяемые классом `Blob`, или метод `stream()` для потокового доступа к содержимому файла.)

При работе с двоичными данными, особенно при потоковой передаче двоичных данных, может требоваться декодирование байтов в текст или кодирование текста в байты. Справиться с задачей помогут классы `TextEncoder` и `TextDecoder`.

15.15.10. API-интерфейсы для работы с медиаданными

Функция `navigator.mediaDevices.getUserMedia()` позволяет коду JavaScript запрашивать доступ к микрофону и/или видеокамере устройства пользователя. Видеопотоки можно отображать в дескрипторе `<video>` (устанавливая свойство `srcObject` в поток). Неподвижные кадры видеопотока можно захватывать в закадровом дескрипторе `<canvas>` с помощью функции `drawImage()` холста, получая в результате фотографию с относительно низким разрешением. Аудио- и видеопотоки, возвращаемые `getUserMedia()`, могут быть записаны и закодированы в `Blob` посредством объекта `MediaRecorder`.

Более сложный API-интерфейс WebRTC делает возможной передачу объектов `MediaStream` по сети, позволяя организовывать, например, одноранговую видеоконференцию.

15.15.11. API-интерфейсы для работы с криптографией и связанные с ними API-интерфейсы

Свойство `crypto` объекта `Window` предоставляет метод `getRandomValues()` для получения криптостойких псевдослучайных чисел. Другие методы для шифрования, расшифровки, генерации ключей, цифровых подписей и т.д. доступны через `crypto.subtle`. Имя свойства (subtle — хитроумное) является предупреждением для всех, кто использует эти методы, что надлежащее применение

криптографических алгоритмов сопряжено с трудностями, и вы должны использовать упомянутые методы, только если по-настоящему понимаете, что делаете. Кроме того, методы `crypto.subtle` доступны лишь коду JavaScript, выполняющемуся внутри документов, которые были загружены через защищенное подключение HTTPS.

API-интерфейсы Credential Management (управление учетными данными) и Web Authentication (веб-аутентификация) позволяют коду JavaScript генерировать, сохранять и восстанавливать открытые ключи (и другие типы) учетных данных и делают возможным создание учетных записей и входных имен без паролей. API-интерфейс для JavaScript состоит в основном из функций `navigator.credentials.create()` и `navigator.credentials.get()`, но чтобы они работали, на стороне сервера требуется солидная инфраструктура. Указанные API-интерфейсы пока еще не поддерживаются повсеместно, но они способны коренным образом изменить способ входа на веб-сайты.

API-интерфейс Payment Request (запрос на оплату) добавляет браузерную поддержку для совершения платежей по кредитным картам через веб-сеть. Он позволяет пользователям безопасно хранить детали, связанные с оплатой, в браузере, так что им не придется вводить номер кредитной карты каждый раз, когда они совершают покупки. Веб-приложения, которым нужно делать запрос на оплату, создают объект `PaymentRequest` и вызывают его метод `show()` для отображения запроса пользователю.

ГЛАВА 16

JavaScript на стороне сервера с использованием Node

Среда Node — это JavaScript с привязками к лежащей в основе операционной системе (ОС), что делает возможным написание программ на JavaScript, которые выполняют чтение и запись файлов, запускают дочерние процессы и взаимодействуют через сеть. В результате среда Node пригодна в качестве:

- современной альтернативы сценариям командной оболочки, которая не обладает загадочным синтаксисом bash и других оболочек Unix;
- универсального языка программирования для выполнения доверенных программ, не подверженного ограничениям безопасности, которые веб-браузеры налагают на ненадежный код;
- популярной среды для написания эффективных веб-серверов с высокой степенью параллелизма.

Определяющей характеристикой среды Node является ее однопоточный основанный на событиях параллелизм, обеспечиваемый по умолчанию асинхронным API-интерфейсом. Если вы программировали на других языках, но не особо много писали код на JavaScript, или если вы — опытный программист на JavaScript клиентской стороны, привыкший писать код для веб-браузеров, то применение Node потребует небольшой подстройки как в случае любого нового языка или среды программирования. Глава начинается объяснением программной модели Node с особым акцентом на параллелизме, API-интерфейсе Node для работы с потоковыми данными и типе `Buffer`, позволяющем иметь дело с двоичными данными в Node. За начальными разделами следуют разделы, где

объясняются и демонстрируются некоторые самые важные API-интерфейсы Node, включая те, что предназначены для работы с файлами, сетями, процессами и потоками.

Одной главы явно недостаточно, чтобы документировать все API-интерфейсы Node, но я надеюсь, что благодаря изложенным в этой главе основам вы получите возможность успешно использовать Node и сумеете освоить любые новые API-интерфейсы, которые вам понадобятся.

Установка Node

Node — программное обеспечение с открытым кодом. Зайдите на веб-сайт `https://nodejs.org`, чтобы загрузить и установить Node для Windows и MacOS. На компьютере с Linux вы можете установить Node с помощью обычного диспетчера пакетов или посетить веб-сайт `https://nodejs.org/en/download` для загрузки двоичных сборок напрямую. Если вы имеете дело с контейнерным программным обеспечением, то можете найти официальные образы Docker для Node по ссылке `https://hub.docker.com`.

Помимо исполняемого файла Node установка Node также включает `npm` — диспетчер пакетов, который обеспечивает легкий доступ к обширной экосистеме инструментов и библиотек JavaScript. В примерах настоящей главы будут применяться только встроенные пакеты Node, а `npm` или любые внешние библиотеки не потребуются.

Наконец, не упускайте из виду официальную документацию по Node, доступную по ссылкам `https://nodejs.org/api` и `https://nodejs.org/docs/guides`. Я обнаружил, что она удобно организована и хорошо написана.

16.1. Основы программирования в Node

В начале главы мы быстро взглянем на то, как программы Node структурированы и каким образом они взаимодействуют с ОС.

16.1.1. Вывод на консоль

Если вы привыкли к программированию на JavaScript для веб-браузеров, тогда одним из небольших сюрпризов, касающихся Node, будет то, что функция `console.log()` предназначена не только для отладки. Она является простейшим способом отображения сообщения пользователю в Node или в более общем плане отправки вывода в поток `stdout`. Вот как выглядит классическая программа "Hello World" в Node:

```
console.log("Hello World!");
```

Имеются и низкоуровневые способы записи в `stdout`, но нет более изящного или официального способа, чем простой вызов `console.log()`.

В веб-браузерах `console.log()`, `console.warn()` и `console.error()`, как правило, отображают маленькие значки рядом со своим выводом в консоли инструментов разработчика, чтобы обозначать различные журнальные сообщения. В Node это не делается, но вывод, который отображается посредством `console.error()`, отличается от вывода, отображаемого с помощью `console.log()`, поскольку `console.error()` осуществляет запись в стандартный поток ошибок `stderr`. Если вы используете Node при написании программы, которая спроектирована для перенаправления `stdout` в файл или канал, тогда можете применять `console.error()`, чтобы отображать текст на консоли, где пользователь будет его видеть, несмотря на то, что текст, выводимый `console.log()`, скрывается.

16.1.2. Аргументы командной строки и переменные среды

Если вы ранее писали программы в стиле Unix, предназначенные для вызова из окна терминала или другого интерфейса командной строки, то вам известно, что такие программы обычно получают свой ввод в первую очередь из аргументов командной строки и во вторую очередь из переменных среды.

Среда Node следует таким соглашениям Unix. Программа Node может читать свои аргументы командной строки из массива строк `process.argv`. Первый элемент этого массива всегда будет путем к исполняемому файлу Node. Второй аргумент — путь к файлу кода JavaScript, который Node выполняет. Оставшиеся элементы в массиве `process.argv` являются аргументами, отделенными друг от друга пробелами, которые вы передали в командной строке, когда вызывали Node. Например, пусть вы сохранили приведенную ниже очень короткую программу Node в файле `argv.js`:

```
console.log(process.argv);
```

Затем вы можете выполнить программу и увидеть примерно такой вывод:

```
$ node --trace-uncaught argv.js --arg1 --arg2 filename
[
  '/usr/local/bin/node',
  '/private/tmp/argv.js',
  '--arg1',
  '--arg2',
  'filename'
]
```

Здесь необходимо отметить два момента.

- Первый и второй элементы `process.argv` будут полными путями в файловой системе к исполняемому файлу Node и запущенному файлу кода JavaScript, даже если вы не вводили их в таком виде.
- Аргументы командной строки, которые предназначены для Node и интерпретируются им, потребляются самим исполняемым файлом Node и не присутствуют в `process.argv`. (Аргумент командной строки `--trace-uncaught` в предыдущем примере фактически не делает ничего

полезного; он нужен лишь для того, чтобы продемонстрировать его отсутствие в выводе.) Любые аргументы (наподобие `--arg1` и `filename`), следующие после имени файла кода JavaScript, будут появляться в `process.argv`.

Программы Node могут также принимать ввод от переменных среды в стиле Unix. Среда Node делает их доступными через объект `process.env`. Именами свойств этого объекта будут имена переменных среды, а значениями (всегда строковыми) свойств — значения переменных среды.

Вот неполный список переменных среды в моей системе:

```
$ node -p -e 'process.env'
{
  SHELL: '/bin/bash',
  USER: 'david',
  PATH: '/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin',
  PWD: '/tmp',
  LANG: 'en_US.UTF-8',
  HOME: '/Users/david',
}
```

Вы можете использовать `node -h` или `node --help`, чтобы выяснить, для чего предназначены аргументы командной строки `-p` и `-e`. Однако имейте в виду, что вы могли бы переписать предыдущую строку следующим образом: `node --eval 'process.env' --print`.

16.1.3. Жизненный цикл программы

Команда `node` ожидает аргумента командной строки, указывающего файл кода JavaScript, который должен быть запущен. Такой начальный файл обычно импортирует другие модули кода JavaScript и может также определять собственные классы и функции. Тем не менее, по существу Node выполняет код JavaScript в заданном файле от начала до конца. Некоторые программы Node прекращают работу, когда завершается выполнение последней строки кода в файле. Однако нередко программа Node будет еще долго функционировать после выполнения начального файла. Как будет обсуждаться в последующих подразделах, программы Node часто являются асинхронными и основанными на обратных вызовах и обработчиках событий. Программы Node не прекращают работу до тех пор, пока не закончится выполнение исходного файла, пока не будут вызваны все обратные вызовы и не будет больше ожидающих событий. Серверная программа, основанная на Node, которая прослушивает входящие сетевые подключения, теоретически будет выполняться вечно, потому что она всегда должна ожидать дополнительных событий.

Программа способна принудительно прекратить свою работу посредством вызова `process.exit()`. Пользователи обычно могут завершать программу Node путем ввода <Ctrl+C> в окне терминала, где программа выполняется. Программа может игнорировать <Ctrl+C>, регистрируя функцию обработчика сигналов с помощью `process.on("SIGINT", ()=>{})`.

Если код в вашей программе генерирует исключение, которое не перехватывается какой-то конструкцией `catch`, тогда программа выведет трассировку стека и прекратит работу. Из-за асинхронной природы Node исключения, которые происходят в обратных вызовах или обработчиках событий, должны обрабатываться локально или вообще не обрабатываться, а это значит, что обработка исключений, возникающих в асинхронных частях вашей программы, может оказаться сложной задачей. Если вы не хотите, чтобы такие исключения приводили к полному отказу вашей программы, тогда зарегистрируйте функцию глобального обработчика, которая будет вызываться взамен отказа:

```
process.setUncaughtExceptionCaptureCallback(e => {
   console.error("Uncaught exception:", e);
                // Неперехваченное исключение
});
```

Аналогичная ситуация возникает, когда созданный вашей программой объект `Promise` отклоняется, но вызова `.catch()` для его обработки не предусмотрено. Начиная с версии Node 13, это не фатальная ошибка, которая приводит к прекращению работы программы, но на консоль выводится развернутое сообщение об ошибке. Ожидается, что в какой-то будущей версии Node необработанные отклонения объектов `Promise` станут фатальными ошибками. Если вы не хотите, чтобы необработанные отклонения приводили к выводу сообщений об ошибках или прекращению работы вашей программы, тогда зарегистрируйте функцию глобального обработчика:

```
process.on("unhandledRejection", (reason, promise) => {
  // reason - любое значение, которое передавалось бы вызову .catch().
  // promise - объект Promise, который был отклонен.
});
```

16.1.4. Модули Node

В главе 10 были описаны системы модулей JavaScript с охватом модулей Node и модулей ES6. Поскольку среда Node была создана до того, как в JavaScript появилась система модулей, в Node пришлось создавать собственную систему такого рода. Система модулей Node применяет функцию `require()` для импортирования значений в модуль и объект `exports` или свойство `module.exports` для экспортирования значений из модуля. Они являются фундаментальной частью программной модели Node и подробно раскрывались в разделе 10.2.

В версии Node 13 добавлена поддержка модулей стандарта ES6, а также модулей, основанных на `require()`, которые в Node называются "модулями CommonJS ". Две системы модулей совместимы не полностью, т.к. сделать это непросто. Перед загрузкой модуля среде Node необходимо знать, будет этот модуль использовать `require()` и `module.exports` или же `import` и `export`. Когда среда Node загружает файл кода JavaScript как модуль CommonJS, она автоматически определяет функцию `require()` наряду с идентификаторами `exports` и `module` и не разрешает применять ключевые слова `import` и `export`. С другой стороны, когда среда загружает файл кода как модуль ES6,

она обязана разрешить объявления import и export и *не* определять добавочные идентификаторы вроде require, module и exports.

Самый простой способ сообщить Node вид загружаемого модуля предусматривает кодирование такой информации в расширении имени файла. Если вы сохраните свой код JavaScript в файле с расширением .mjs, то Node будет всегда загружать его как модуль ES6, ожидать в нем использования import и export и не предоставлять функцию require(). Если же вы сохраните свой код в файле с расширением .cjs, тогда Node будет всегда обходиться с ним как с модулем CommonJS, предоставлять функцию require() и генерировать SyntaxError в случае применения объявлений import или export.

Для файлов, которые не имеют явного расширения .mjs или .cjs, среда Node ищет файл по имени package.json в том же самом каталоге и затем в содержащихся подкаталогах. После того, как найден ближайший файл package.json, среда Node проверяет свойство type верхнего уровня в объекте JSON. Если значением свойства type оказывается "module", то Node загрузит файл как модуль ES6, а если значением type является "commonjs", тогда Node загрузит файл как модуль CommonJS. Обратите внимание, что для запуска программ Node иметь файл package.json вовсе не обязательно: когда такой файл не найден (или он найден, но не содержит свойство type), то Node по умолчанию использует модули CommonJS. Прием с файлом package.json становится необходимым, только если вы хотите применять модули ES6 с Node и не желаете использовать расширение имени файла .mjs.

Поскольку существует огромный объем кода Node, написанного с применением формата модулей CommonJS, среда Node разрешает модулям ES6 загружать модули CommonJS, используя ключевое слово import. Тем не менее, обратное утверждение неверно: модуль CommonJS не может применять require() для загрузки модуля ES6.

16.1.5. Диспетчер пакетов Node

В результате установки Node вы обычно получаете также программу по имени npm. Это диспетчер пакетов Node (Node Package Manager), помогающий загружать и управлять библиотеками, от которых зависит ваша программа. Диспетчер npm отслеживает такие зависимости (и другую информацию о вашей программе) в файле package.json, расположенном в корневом каталоге проекта. Именно в создаваемый диспетчером npm файл package.json вы добавляли бы "type":"module" при желании использовать модули ES6 для своего проекта.

В настоящей главе диспетчер npm подробно не рассматривается (но чуть больше сведений можно найти в разделе 17.4). Я упоминаю о нем здесь из-за того, что если только вы не пишете программы, которые вообще не задействуют внешние библиотеки, то почти наверняка будете применять npm или похожий инструмент. Предположим, например, что вы собираетесь разработать веб-сервер и для упрощения задачи планируете использовать фреймворк Express (https://expressjs.com). Первым делом вы могли бы создать каталог для

своего проекта и затем ввести в этом каталоге команду npm init. Диспетчер npm запросит у вас имя проекта, номер версии и т.д., после чего на основе ваших ответов создаст начальный файл package.json.

Теперь, чтобы приступить к применению Express, вы ввели бы команду npm install express, которая сообщает диспетчеру npm о необходимости загрузить библиотеку Express вместе со всеми ее зависимостями и установить все пакеты в локальный каталог node_modules/:

```
$ npm install express
npm notice created a lockfile as package-lock.json.
You should commit this file.
npm WARN my-server@1.0.0 No description
npm WARN my-server@1.0.0 No repository field.
npm заметил созданный файл package-lock.json.
Вы должны зафиксировать этот файл.
npm WARN my-server@1.0.0 Нет описания
npm WARN my-server@1.0.0 Нет поля хранилища.

+ express@4.17.1
added 50 packages from 37 contributors and audited 126 packages in 3.058s
found 0 vulnerabilities
добавлено 50 пакетов от 37 участников и проверено 126 пакетов
за 3.058 с обнаружено 0 уязвимостей
```

Когда вы устанавливаете пакет с помощью npm, диспетчер npm записывает такую зависимость — зависимость вашего проекта от Express — в файл package.json. Благодаря записи этой зависимости package.json вы можете предоставить другому программисту копию своего кода и файл package.json, а тому придется просто ввести команду npm install для автоматической загрузки и установки всех библиотек, которые нужны для запуска вашей программы.

16.2. Среда Node асинхронна по умолчанию

JavaScript — универсальный язык программирования, поэтому вполне возможно писать программы, интенсивно использующие центральный процессор (ЦП), которые перемножают крупные матрицы или выполняют сложный статистический анализ. Но среда Node была спроектирована и оптимизирована для программ с интенсивным вводом-выводом, подобных сетевым серверам. В частности, среда Node конструировалась для того, чтобы можно было легко реализовывать серверы с высокой степенью параллелизма, которые способны обрабатывать множество запросов одновременно.

Однако в отличие от многих языков программирования Node не добивается параллелизма посредством потоков. Как известно, многопоточный код трудно писать правильно и сложно отлаживать. Кроме того, потоки представляют собой относительно тяжеловесную абстракцию, и если вы хотите реализовать сервер, который может обрабатывать сотни параллельных запросов, применение сотен потоков может требовать расхода чрезмерно высокого объема памяти.

Таким образом, Node принимает однопоточную программную модель JavaScript, используемую в веб-сети, что в свою очередь оказывается огромным упрощением, которое делает создание сетевых серверов обыкновенным навыком, а не загадочным искусством.

Подлинный параллелизм с помощью Node

Программы Node могут запускать множество процессов ОС, а Node 10 и последующие версии поддерживают объекты `Worker` (см. раздел 16.11), которые являются разновидностью потоков, позаимствованных у веб-браузеров. Если вы применяете множество процессов или создаете один или большее количество потоков `Worker` и запускаете свою программу в системе, имеющей более одного ЦП, тогда программа больше не будет однопоточной и станет по-настоящему выполнять множество потов кода параллельно. Такие методики могут быть полезны для операций, интенсивно использующих ЦП, но обычно не применяются для программ с интенсивным вводом-выводом вроде серверов.

Тем не менее, стоит отметить, что процессы и потоки `Worker` среды Node избегают типичной сложности многопоточного программирования, потому что взаимодействие между процессами и между потоками `Worker` осуществляется через передачу сообщений, и они не позволяют легко совместно использовать память друг с другом.

Среда Node добивается высоких уровней параллелизма и в то же время сохраняет однопоточную программную модель за счет того, что делает собственный API-интерфейс асинхронным и неблокирующим по умолчанию. Среда Node очень серьезно относится к своему неблокирующему подходу и проявляет крайности, которые могут вас удивить. Вероятно, вы ожидаете, что функции, которые выполняют чтение и запись в сеть, будут асинхронными, но Node идет дальше и определяет неблокирующие асинхронные функции для чтения и записи файлов в локальной файловой системе. Если подумать, то это имеет смысл: API-интерфейс Node проектировался в те дни, когда вращающиеся жесткие диски были нормой и действительно проходили миллисекунды блокирующего "времени позиционирования" в ожидании вращения диска, прежде чем файловая операция могла начаться. А в современных центрах обработки данных "локальная" файловая система на самом деле может располагаться где-то в сети, что добавляет к задержкам дисков еще и сетевые задержки. Но даже если чтение файла асинхронным образом кажется вам нормальным, Node продвигается еще дальше: стандартные функции, скажем, для инициирования сетевого подключения или поиска времени изменения файла также реализованы как неблокирующие.

Некоторые функции в API-интерфейсе Node являются синхронными, но неблокирующими: они выполняются до завершения и возвращают управление без потребности в блокировке. Но большинство интересных функций выполняют ввод или вывод какого-то вида и они асинхронные, так что могут избежать даже

крошечной доли блокирования. Среда Node создавалась до того, как в JavaScript появился класс Promise, и потому асинхронные API-интерфейсы Node основаны на обратных вызовах. (Если вы еще не читали или уже забыли материал главы 13, то самое время возвратиться к ней.) Как правило, последним аргументом, передаваемым асинхронной функции Node, будет обратный вызов. В Node применяются *обратные вызовы с первым аргументом-ошибкой* (error-first callback), которые обычно вызываются с двумя аргументами. В первом аргументе такого обратного вызова содержится null в случае, когда никаких ошибок не возникало, а во втором аргументе — любые данные или ответ, произведенные исходной асинхронной функцией, которую вы вызвали. Причина помещения аргумента-ошибки первым связана с тем, чтобы его нельзя было опустить, и вы должны всегда проверять его значение на предмет неравенства null. Если он представляет собой объект Error либо даже целочисленный код ошибки или сообщение об ошибке, тогда что-то пошло не так. В таком случае второй аргумент функции обратного вызова, скорее всего, будет равен null.

В следующем коде показано, как использовать неблокирующую функцию readFile() для чтения файла конфигурации, его разбора в виде JSON и передачи разобранного объекта конфигурации другому обратному вызову:

```
const fs = require("fs");   // Затребовать модуль для работы
                            // с файловой системой.

// Читает файл конфигурации, разбирает его содержимое как JSON
// и передает результирующее значение обратному вызову.
// Если что-то идет не так, тогда выводит сообщение об ошибке
// в stderr и вызывает обратный вызов с null.
function readConfigFile(path, callback) {
   fs.readFile(path, "utf8", (err, text) => {
      if (err) {            // Что-то пошло не так при чтении файла.
         console.error(err);
         callback(null);
         return;
      }
      let data = null;
      try {
         data = JSON.parse(text);
      } catch(e) {//Что-то пошло не так при разборе содержимого файла.
         console.error(e);
      }
      callback(data);
   });
}
```

Среда Node предшествует стандартизированным объектам Promise, но поскольку она весьма согласована в отношении своих обратных вызовов с первым аргументом-ошибкой, довольно легко создать основанные на Promise варианты ее API-интерфейсов на базе обратных вызовов с применением оболочки util.promisify(). Вот как можно было бы переписать функцию readConfigFile() для возвращения объекта Promise:

```
const util = require("util");
const fs = require("fs");   // Затребовать модуль для работы
                            // с файловой системой.
const pfs = {  // Основанные на Promise варианты некоторых функций fs.
   readFile: util.promisify(fs.readFile)
};
function readConfigFile(path) {
   return pfs.readFile(path, "utf-8").then(text => {
      return JSON.parse(text);
   });
}
```

Мы также можем упростить предыдущую функцию на основе Promise, используя async и await (и снова, если вы еще не читали главу 13, то займитесь этим сейчас):

```
async function readConfigFile(path) {
   let text = await pfs.readFile(path, "utf-8");
   return JSON.parse(text);
}
```

Оболочка util.promisify() может создавать основанные на Promise версии многих функций Node. В Node 10 и последующих версиях объект fs.promises имеет несколько предопределенных функций на основе Promise для работы с файловой системой. Мы обсудим их позже в главе, но имейте в виду, что в предыдущем коде мы могли бы заменить pfs.readFile() вызовом fs.promises.readFile().

Ранее уже было указано, что программная модель Node асинхронна по умолчанию. Но для удобства программистов Node определяет блокирующие синхронные варианты многих своих функций, особенно в модуле для работы с файловой системой. Такие функции обычно имеют имена, явно помеченные с помощью Sync в конце.

Когда сервер запускается впервые и читает свои файлы конфигурации, он еще не обрабатывает сетевые запросы и потому возможен лишь небольшой параллелизм или вообще никакой. Таким образом, в этой ситуации действительно нет необходимости избегать блокирования, и мы можем безопасно применять блокирующие функции вроде fs.readFileSync(). В этом коде мы можем отказаться от async и await и написать чисто синхронную версию нашей функции readConfigFile(). Вместо того чтобы вызывать обратный вызов или возвращать объект Promise, данная функция просто возвращает разобранное значение JSON или генерирует исключение:

```
const fs = require("fs");
function readConfigFileSync(path) {
   let text = fs.readFileSync(path, "utf-8");
   return JSON.parse(text);
}
```

Помимо обратных вызовов с первым аргументом-ошибкой, принимающих два аргумента, в Node также имеется несколько API-интерфейсов, которые ис-

пользуют асинхронность, основанную на событиях, обычно для обработки потоковых данных. Мы рассмотрим события Node более подробно позже в главе.

Теперь, обсудив активно неблокирующий API-интерфейс Node, давайте вернемся к теме параллелизма. Встроенные неблокирующие функции Node работают с применением версий обратных вызовов и обработчиков событий, предоставляемых ОС. Когда вы вызываете одну из этих функций, Node предпринимает действие для запуска операции, а затем регистрирует в ОС обработчик событий определенного вида, чтобы получить уведомление о завершении операции. Обратный вызов, который вы передаете функции Node, сохраняется внутренне, так что среда Node может вызвать ваш обратный вызов, когда ОС отправит ей соответствующее сообщение.

Такую разновидность параллелизма часто называют параллелизмом, основанным на событиях. В своей основе Node имеет единственный поток, который выполняет "цикл обработки событий". При запуске программа Node выполняет любой код, о необходимости выполнения которого вы ей сообщили. Этот код предположительно вызывает, по меньшей мере, одну неблокирующую функцию, приводящую к регистрации обратного вызова или обработчика событий в ОС. (Если нет, тогда вы написали синхронную программу Node и среда Node просто прекращает работу, добравшись до ее конца.) Когда среда Node достигает конца вашей программы, она блокируется до тех пор, пока не произойдет событие, после чего ОС возобновит ее работу. Среда Node сопоставляет событие ОС с зарегистрированным вами обратным вызовом JavaScript и затем вызывает функцию обратного вызова. Ваша функция обратного вызова может вызывать дополнительные неблокирующие функции Node, что приводит к регистрации добавочных обработчиков событий ОС. Как только ваша функция обратного вызова завершает выполнение, Node возвращается в спящий режим и цикл повторяется.

Для веб-серверов и других приложений с интенсивным вводом-выводом, которые проводят *большую* часть своего времени в ожидании ввода и вывода, такой стиль параллелизма на основе событий будет эффективным и действенным. Веб-сервер может параллельно обрабатывать запросы от 50 разных клиентов, не требуя 50 разных потоков, если он использует неблокирующие API-интерфейсы и существует какое-то внутреннее сопоставление сетевых сокетов с функциями JavaScript, которые должны вызываться при возникновении активности в этих сокетах.

16.3. Буферы

Одним из типов данных, с которым вам, вероятно, часто придется иметь дело в Node, особенно при чтении данных из файлов либо из сети, является класс `Buffer`. Буфер во многом похож на строку, но вместо последовательности символов содержит последовательность байтов. Среда Node создавалась до того, как в базовом JavaScript стали поддерживаться типизированные массивы (см. раздел 11.2), и не было типа `Uint8Array` для представления массива байтов без знака. Класс `Buffer` определен в Node именно для того, чтобы удовлетворить

такую потребность. Теперь, когда Uint8Array входит в состав языка JavaScript, класс Buffer среды Node реализован как подкласс Uint8Array.

Класс Buffer отличается от своего суперкласса Uint8Array тем, что он спроектирован для взаимодействия со строками JavaScript: байты в буфере могут быть инициализированы из символьных строк и преобразованы в символьные строки. Кодировка символов сопоставляет каждый символ в некотором наборе символов с целым числом. Имея строку текста и кодировку символов, мы можем *закодировать* символы строки в последовательность байтов. И располагая (надлежащим образом закодированной) последовательностью байтов и кодировкой символов, мы можем *декодировать* байты в последовательность символов. Класс Buffer среды Node имеет методы, которые выполняют кодирование и декодирование, и вы можете их легко опознать, потому что они ожидают аргумент encoding, указывающий применяемую кодировку.

Кодировки в Node задаются по имени в виде строки. Ниже перечислены поддерживаемые кодировки.

- "utf8". Принимается по умолчанию, когда кодировка не указана, и является кодировкой Unicode, которую вы наиболее вероятно будете использовать.
- "utf16le". Двухбайтовые символы Unicode с порядком байтов от младшего к старшему. Кодовые точки выше \uffff кодируются как пары двухбайтовых последовательностей. Кодировка "ucs2" представляет собой псевдоним.
- "latin1". Кодировка ISO-8859-1 с одним байтом на символ, которая определяет набор символов, подходящий для многих западноевропейских языков. Поскольку существует однозначное соответствие между байтами и символами latin-1, эта кодировка также известна как "binary".
- "ascii". 7-битная кодировка ASCII только для английского языка, которая является строгим подмножеством кодировки "utf8".
- "hex". Эта кодировка преобразует каждый байт в пару шестнадцатеричных цифр ASCII.
- "base64". Эта кодировка преобразует каждую последовательность из трех байтов в последовательность из четырех символов ASCII.

В следующем примере кода показано, как работать с буферами и преобразовывать их в строки и из строк:

```
let b = Buffer.from([0x41, 0x42, 0x43]);     // <Buffer 41 42 43>
b.toString()                          // => "ABC"; по умолчанию "utf8"
b.toString("hex")                             // => "414243"

let computer=Buffer.from("IBM3111","ascii"); // Преобразование
                                              // строки в буфер.
for(let i = 0; i < computer.length; i++) {    // Использование буфера
                                              // как байтового массива.
   computer[i]--;                             // Буферы изменяемы.
}
```

```
computer.toString("ascii")                          // => "HAL2000"
computer.subarray(0,3).map(x=>x+1).toString()       // => "IBM"

// Создать новые "пустые" буферы с помощью Buffer.alloc().
let zeros = Buffer.alloc(1024);                     // 1024 нуля.
let ones = Buffer.alloc(128, 1);                    // 128 единиц.
let dead = Buffer.alloc(1024, "DEADBEEF", "hex");   // Повторяющийся
                                                    // шаблон байтов.
// Буферы имеют методы для чтения и записи многобайтовых
// значений из и в буфер с любым указанным смещением.
dead.readUInt32BE(0)        // => 0xDEADBEEF
dead.readUInt32BE(1)        // => 0xADBEEFDE
dead.readBigUInt64BE(6)     // => 0xBEEFDEADBEEFDEADn
dead.readUInt32LE(1020)     // => 0xEFBEADDE
```

Если вы пишете программу Node, которая фактически манипулирует двоичными данными, то можете обнаружить, что широко применяете класс `Buffer`. С другой стороны, если вы просто работаете с текстом, который читается или записывается в файл или в сеть, тогда можете встретить класс `Buffer` только в качестве промежуточного представления ваших данных. Некоторые API-интерфейсы Node способны принимать ввод или возвращать вывод в виде либо строк, либо объектов `Buffer`. Обычно если вы передаете строку одному API-интерфейсу такого рода или ожидаете, что строка будет им возвращена, то должны указывать имя желаемой кодировки текста. И если вы это делаете, тогда использовать объект `Buffer` вообще не придется.

16.4. События и EventEmitter

Как неоднократно упоминалось ранее, все API-интерфейсы Node асинхронны по умолчанию. Для многих из них такая асинхронность имеет форму обратных вызовов с первым аргументом-ошибкой, принимающих два аргумента, которые вызываются, когда запрошенная операция завершена. Но некоторые более сложные API-интерфейсы взамен основаны на событиях. Обычно так происходит в случае, когда API-интерфейс разработан относительно объекта, а не функции, когда функция обратного вызова должна вызываться много раз или когда может требоваться множество типов функций обратного вызова. В качестве примера возьмем класс `net.Server`: объект этого типа представляет серверный сокет, который применяется для приема входящих подключений от клиентов. Он выпускает событие `"listening"`, когда начинает прослушивание подключений, событие `"connection"` при каждом подключении клиента и событие `"close"`, когда он был закрыт и больше не прослушивает подключения.

В среде Node объекты, которые выпускают события, являются экземплярами класса `EventEmitter` или какого-то подкласса `EventEmitter`:

```
const EventEmitter = require("events"); // Имя модуля не совпадает
                                        // с именем класса.
const net = require("net");
let server = new net.Server();  // Создать объект Server.
server instanceof EventEmitter  // => true: объекты Server являются
                                //    экземплярами EventEmitter.
```

Главная особенность экземпляров EventEmitter заключается в том, что они позволяют регистрировать функции обработчиков событий с помощью метода on(). Экземпляры EventEmitter могут выпускать множество типов событий, а типы событий идентифицируются по именам. Чтобы зарегистрировать обработчик событий, вызовите метод on(), передав ему имя типа событий и функцию, которая должна вызываться, когда происходит событие указанного типа. Экземпляры EventEmitter могут вызывать функции обработчиков с любым количеством аргументов, и чтобы узнать, передачу каких аргументов следует ожидать, вам придется почитать документацию по специфическому виду событий, поступающих от специфического экземпляра EventEmitter:

```
const net = require("net");
let server = new net.Server();      //Создать объект Server.
server.on("connection", socket => { //Прослушивать события "connection".
    // События "connection" объекта Server передаются объекту socket
    // для только что подключившегося клиента. Здесь мы посылаем
    // клиенту определенные данные и отключаемся.
    socket.end("Hello World", "utf8");
});
```

Если вы предпочитаете более явные имена методов для регистрации прослушивателей событий, то можете также использовать метод addListener(). Кроме того, вы можете удалять ранее зарегистрированный прослушиватель событий посредством метода off() или removeListener(). В качестве особого случая вы можете зарегистрировать прослушиватель событий, который будет автоматически удаляться после его первого запуска, вызвав метод once() вместо on().

При возникновении события определенного типа в определенном объекте EventEmitter среда Node вызывает все функции обработчиков, которые в текущий момент зарегистрированы в данном объекте EventEmitter для событий такого типа. Они вызываются в порядке от первого зарегистрированного до последнего зарегистрированного. Если есть более одной функции обработчика, тогда они вызываются последовательно в одном потоке: не забывайте, что в Node отсутствует параллелизм. И, что важно, функции обработчиков событий вызываются синхронно, а не асинхронно. Таким образом, метод emit() не помещает обработчики событий в очередь для вызова в более позднее время. Метод emit() вызывает все зарегистрированные обработчики друг за другом и не возвращает управление до тех пор, пока не возвратит управление последний обработчик событий.

Фактически это означает, что когда один из встроенных API-интерфейсов Node выпускает событие, то данный API-интерфейс по существу блокирует ваши обработчики событий. Если вы напишете обработчик событий, в котором вызывается блокирующая функция вроде fs.readFileSync(), то дальнейшая обработка событий не будет происходить, пока не завершится синхронное чтение файла. Если ваша программа должна быть отзывчивой подобно сетевому серверу, тогда важно, чтобы функции обработчиков событий были неблокирующими и быстрыми. Если вам необходимо выполнять много вычислений, когда возникает событие, то часто лучше применять обработчик для планирования этих вычислений асинхронным образом, используя setTimeout() (см. раздел 11.10).

В Node также определена функция `setImmediate()`, которая планирует вызов указанной функции немедленно после обработки всех ожидающих обратных вызовов и событий.

В классе `EventEmitter` также определен метод `emit()`, который обеспечивает вызов зарегистрированных функций обработчиков событий. Он полезен, если вы определяете собственный API-интерфейс на основе событий, но обычно не применяется при программировании с использованием существующих API-интерфейсов. Метод `emit()` должен вызываться с именем типа событий в первом аргументе. Любые дополнительные аргументы, передаваемые `emit()`, становятся аргументами для зарегистрированных функций обработчиков событий. Функции обработчиков также вызываются со значением `this`, установленным в сам объект `EventEmitter`, что часто удобно. (Однако не забывайте, что стрелочные функции всегда применяют значение `this` контекста, в котором они определены, и не могут вызываться с любым другим значением `this`. Тем не менее, стрелочные функции часто оказываются наиболее удобным способом написания обработчиков событий.)

Любое значение, возвращаемое функцией обработчика событий, игнорируется. Однако если функция обработчика событий генерирует исключение, то оно распространяется из вызова `emit()` и препятствует выполнению любых функций обработчиков, которые были зарегистрированы после той, что сгенерировала исключение.

Вспомните, что основанные на обратных вызовах API-интерфейсы Node используют обратные вызовы с первым аргументом-ошибкой, и важно всегда проверять первый аргумент обратного вызова, чтобы выяснить, не возникала ли ошибка. В API-интерфейсах на основе событий эквивалентом являются события `"error"`. Так как API-интерфейсы на основе событий часто применяются для взаимодействия с сетью и других форм потокового ввода-вывода, они уязвимы к непредсказуемым асинхронным ошибкам, и в большинстве классов `EventEmitter` определено событие `"error"`, которое инициируется при возникновении ошибки. Всякий раз, когда вы используете API-интерфейс на основе событий, нужно выработать у себя привычку регистрировать обработчик для событий `"error"`. События `"error"` в классе `EventEmitter` трактуются особым образом. Если метод `emit()` вызывается для выпуска события `"error"` и если для этого типа событий не были зарегистрированы обработчики, тогда сгенерируется исключение. Поскольку подобное происходит асинхронно, нет никакой возможности обработать исключение в блоке `catch`, а потому такой вид ошибки обычно приводит к завершению работы программы.

16.5. Потоки данных

При реализации алгоритма для обработки данных почти всегда легче всего прочитать все данные в память, обработать их и затем записать готовые данные. Скажем, вот как вы могли бы реализовать функцию Node для копирования файла[1].

[1] В Node определена функция `fs.copyFile()`, которую вы применяли бы на практике.

```
const fs = require("fs");

// Асинхронная, но не потоковая (и потому неэффективная) функция.
function copyFile(sourceFilename, destinationFilename, callback) {
    fs.readFile(sourceFilename, (err, buffer) => {
        if (err) {
            callback(err);
        } else {
            fs.writeFile(destinationFilename, buffer, callback);
        }
    });
}
```

В функции `copyFile()` используются асинхронные функции и обратные вызовы, так что она не блокируется и подходит для применения в параллельных программах, подобных серверам. Но обратите внимание, что она обязана выделять достаточно памяти для хранения полного содержимого файла. В ряде случаев такой подход вполне нормален, но он начинает терпеть неудачу, если копируемые файлы очень велики или если ваша программа обладает высокой степенью параллелизма и может одновременно копировать множество файлов. Еще один недостаток показанной реализации `copyFile()` заключается в том, что она не может начать запись нового файла, пока не завершит чтение старого файла.

Решение указанных проблем предусматривает использование потоковых алгоритмов, где данные "затекают" в вашу программу, обрабатываются и "вытекают" из нее. Идея в том, что ваш алгоритм обрабатывает данные небольшими порциями и полный набор данных никогда не хранится в памяти целиком. Когда потоковые решения возможны, они более эффективны с точки зрения памяти и также могут быть более быстрыми. API-интерфейсы взаимодействия с сетью Node основаны на потоках и в модуле для работы с файловой системой Node определены потоковые API-интерфейсы чтения и записи файлов, поэтому вы, вероятно, применяли какой-нибудь потоковый API-интерфейс во многих программах Node, которые вам приходилось реализовывать. Вы увидите потоковую версию функции `copyFile()` в подразделе "Режим извлечения данных" далее в главе.

В Node поддерживаются четыре основных типа потоков.

- **`Readable`** (допускающий чтение). Потоки `Readable` являются источниками данных. Например, поток, возвращаемый `fs.createReadStream()`, представляет собой поток, из которого можно читать содержимое указанного файла. `process.stdin` — еще один поток `Readable`, который возвращает данные из стандартного ввода.
- **`Writable`** (допускающий запись). Потоки `Writable` являются приемниками или пунктами назначения для данных. Скажем, возвращаемое значение `fs.createWriteStream()` — это поток `Writable`: он позволяет записывать данные по частям и выводит все записанные данные в указанный файл.

- **Duplex** (дуплексный). Потоки Duplex объединяют поток Readable и поток Writable в один объект. Например, объекты Socket, возвращаемые net.connect() и другими API-интерфейсами взаимодействия с сетью Node, являются потоками Duplex. Если вы записываете в сокет, то ваши данные посылаются по сети компьютеру, к которому подключен сокет. А если вы читаете из сокета, то получаете доступ к данным, записанным на этом другом компьютере.
- **Transform** (видоизменяющий). Потоки Transform тоже допускают чтение и запись, но отличаются от потоков Duplex одним важным аспектом: данные, записанные в поток Transform, становятся читабельными — обычно в видоизмененной форме — из того же самого потока. Скажем, функция zlib.createGzip() возвращает поток Transform, который сжимает (с помощью алгоритма *gzip*) записываемые в него данные. Аналогичным образом функция crypto.createCipheriv() возвращает поток Transform, шифрующий или дешифрующий данные, которые в него записываются.

По умолчанию потоки читают и записывают буферы. Если вы вызовите метод setEncoding() потока Readable, то он будет возвращать декодированные строки вам, а нет объектам Buffer. И если вы запишете строку в буфер потока Writable, тогда она будет автоматически закодирована с использованием стандартной кодировки буфера или любой кодировки, которую вы укажете. API-интерфейс потоков Node также поддерживает "объектный режим", при котором потоки читают и записывают объекты, более сложные, чем буферы и строки. Ни один из базовых API-интерфейсов Node не применяет такой объектный режим, но вы можете встретить его в других библиотеках.

Потоки Readable должны откуда-то читать свои данные, а потоки Writable должны куда-то записывать свои данные, и потому каждый поток имеет два конца: ввод и вывод либо источник и назначение. Сложность основанных на потоках API-интерфейсов связана с тем, что данные на двух концах потока почти всегда протекают с разными скоростями. Возможно код, который читает из потока, желает читать и обрабатывать данные быстрее, чем данные фактически записываются в поток. Или наоборот: возможно данные записываются в поток быстрее, чем их можно прочитать и извлечь из потока на другом конце. Реализации потоков практически всегда включают внутренний буфер для удержания данных, которые были записаны, но еще не прочитаны. Буферизация помогает обеспечивать то, что есть данные, доступные для чтения, когда они запрашиваются, и есть пространство для хранения данных, когда они записываются. Но ни то, ни другое никогда не может быть гарантированным, и природа программирования на основе потоков такова, что объектам чтения временами придется ожидать, пока данные будут записаны (т.к. буфер потока пуст), а объектам записи иногда приходится ждать, когда данные будут прочитаны (т.к. буфер потока полон).

В программных средах, которые используют основанный на потоках параллелизм, API-интерфейсы потоков обычно имеют блокирующие вызовы: вызов для чтения данных не возвращает управление до тех пор, пока данные не поступят

в поток, а вызов для записи данных блокируется до тех пор, пока во внутреннем буфере потока не будет достаточно пространства, чтобы разместить новые данные. Тем не менее, в модели параллелизма на основе событий блокирующие вызовы не имеют смысла, и API-интерфейсы потоков Node основаны на событиях и обратных вызовах. В отличие от других API-интерфейсов не существуют синхронные версии методов, которые будут описаны позже в главе.

Необходимость координирования способности к чтению (буфер не пуст) и к записи (буфер не полон) потоков через события несколько усложняет API-интерфейсы потоков Node. Положение усугубляется тем фактом, что эти API-интерфейсы с годами развивались и изменялись: для потоков `Readable` есть два совершенно разных API-интерфейса, которые вы можете применять. Несмотря на сложность, потоковые API-интерфейсы Node стоит понять и освоить, потому что они делают возможным высокопроизводительный ввод-вывод в ваших программах.

В последующих подразделах демонстрируется чтение и запись из классов потоков Node.

16.5.1. Каналы

Иногда вам нужно прочитать данные из потока всего лишь для того, чтобы записать те же самые данные в другой поток. Предположим, например, что вы реализуете простой HTTP-сервер, который обслуживает каталог статических файлов. В таком случае вам понадобится читать данные из файлового потока ввода и записывать их в сетевой сокет. Но вместо написания собственного кода для поддержки чтения и записи вы можете просто соединить два потока вместе в виде "канала" и позволить Node самостоятельно справиться со всеми сложностями. Просто передайте поток `Writable` методу `pipe()` потока `Readable`:

```
const fs = require("fs");
function pipeFileToSocket(filename, socket) {
    fs.createReadStream(filename).pipe(socket);
}
```

Следующая служебная функция создает канал между двумя потоками и вызывает обратный вызов, когда все готово или возникла ошибка:

```
function pipe(readable, writable, callback) {
    // Сначала настроить обработку ошибок.
    function handleError(err) {
        readable.close();
        writable.close();
        callback(err);
    }
    // Затем определить канал и обработать случай нормального завершения.
    readable
        .on("error", handleError)
        .pipe(writable)
        .on("error", handleError)
        .on("finish", callback);
}
```

Потоки Transform особенно удобны с каналами и создают каналы, в которые вовлечено более двух потоков. Вот пример функции, которая сжимает файл:

```
const fs = require("fs");
const zlib = require("zlib");

function gzip(filename, callback) {
    // Создать потоки.
    let source = fs.createReadStream(filename);
    let destination = fs.createWriteStream(filename + ".gz");
    let gzipper = zlib.createGzip();

    // Настроить канал.
    source
        .on("error", callback)    // Вызвать callback при ошибке чтения.
        .pipe(gzipper)
        .pipe(destination)
        .on("error", callback)    // Вызвать callback при ошибке записи.
        .on("finish", callback); // Вызвать callback, когда запись
                                 // завершена.
}
```

Использовать метод pipe() для копирования данных из потока Readable в поток Writable легко, но на практике вам часто необходимо каким-то образом обрабатывать данные по мере их прохождения через вашу программу. Один из способов сделать это предусматривает реализацию собственного потока Transform для выполнения нужной обработки, и такой подход позволяет избежать чтения и записи потоков вручную. Скажем, ниже приведена функция, которая работает подобно утилите grep в Unix: она читает строки текста из входного потока, но записывает только те строки, которые соответствуют указанному регулярному выражению:

```
const stream = require("stream");

class GrepStream extends stream.Transform {
    constructor(pattern) {
      super({decodeStrings:false});// Не преобразовывать строки
                                   // в буферы.
      this.pattern = pattern;      // Регулярное выражение для
                                   // сопоставления.
      this.incompleteLine = "";    // Остаток последней порции данных.
    }

    // Этот метод вызывается, когда есть строка, готовая
    // к видоизменению. Он должен передать видоизмененные данные
    // указанной функции обратного вызова. Мы ожидаем строковый ввод,
    // поэтому данный поток должен подключаться только к потокам
    // Readable, для которых был вызван метод setEncoding().
    _transform(chunk, encoding, callback) {
        if (typeof chunk !== "string") {
            callback(new Error("Expected a string but got a buffer"));
                                // Ожидается строка, но получен буфер
            return;
        }
```

```
        // Добавить chunk к любой ранее незавершенной строке
        // и разбить все на строки.
        let lines = (this.incompleteLine + chunk).split("\n");

        // Последний элемент массива - это новая незавершенная строка.
        this.incompleteLine = lines.pop();

        // Найти все соответствующие строки.
        let output = lines   // Начать со всех завершенных строк,
            .filter(l => this.pattern.test(l))  // отфильтровать из них
            .join("\n");     // совпадения, и снова соединить вместе.

        // Если хоть что-то совпало, тогда добавить финальный
        // символ новой строки.
        if (output) {
            output += "\n";
        }
        // Всегда вызывать обратный вызов, даже если нет никакого вывода
        callback(null, output);
    }
    // Это вызывается сразу после закрытия потока.
    // Мы получаем шанс записать последние данные.
    _flush(callback) {
        // Если все еще есть незавершенная строка, которая
        // дает совпадение, передать ее обратному вызову.
        if (this.pattern.test(this.incompleteLine)) {
            callback(null, this.incompleteLine + "\n");
        }
    }
}
// Теперь с помощью класса GrepStream мы можем написать
// программу наподобие grep.
let pattern = new RegExp(process.argv[2]);  // Получить регулярное
                                            // выражение из командной строки.
process.stdin                               // Начать со стандартного ввода,
    .setEncoding("utf8")                    // читать его как строки Unicode,
    .pipe(new GrepStream(pattern))          // соединить его каналом с
    .pipe(process.stdout)                   // GrepStream и далее соединить
                                            // каналом со стандартным выводом.
    .on("error", () => process.exit());     // Элегантно завершить работу,
                                            // если stdout закрывается.
```

16.5.2. Асинхронная итерация

В Node 12 и последующих версиях потоки `Readable` являются асинхронными итераторами, т.е. внутри функции `async` можно применять цикл `for/await` для чтения строки и порций буфера из потока, используя код, который структурирован как синхронный код. (Дополнительные сведения по асинхронным итераторам и циклам `for/await` приведены в разделе 13.4.)

Применять асинхронный итератор почти так же просто, как метод `pipe()`, и вероятно проще, когда каждую читаемую порцию данных нужно каким-то образом обрабатывать. Вот как мы могли бы переписать программу из предыдущего подраздела с использованием функции `async` и цикла `for/await`:

```
// Читает строки текста из потока источника и записывает любые строки,
// которые соответствуют указанному шаблону, в поток назначения.
async function grep(source, destination, pattern, encoding="utf8") {
    // Настроить поток источника для чтения строк, а не буферов.
    source.setEncoding(encoding);

    // Настроить обработчик ошибок для потока назначения на случай
    // непредвиденного закрытия стандартного вывода.
    destination.on("error", err => process.exit());

    // Читаемые порции вряд ли заканчиваются символом новой строки,
    // поэтому в конце каждой порции, вероятно, будет неполная строка.
    // Отслеживать это здесь.
    let incompleteLine = "";

    // Использовать цикл for/await для асинхронного
    // чтения порций из потока ввода.
    for await (let chunk of source) {
        // Разбить конец последней и этой порции на строки.
        let lines = (incompleteLine + chunk).split("\n");
        // Последняя строка неполная.
        incompleteLine = lines.pop();
        // Пройти в цикле по строкам и записать
        // найденные совпадения в поток назначения.
        for(let line of lines) {
            if (pattern.test(line)) {
                destination.write(line + "\n", encoding);
            }
        }
    }
    // Наконец, проверить на предмет совпадения хвостовой текст.
    if (pattern.test(incompleteLine)) {
        destination.write(incompleteLine + "\n", encoding);
    }
}
let pattern = new RegExp(process.argv[2]);    // Получить регулярное
                                              // выражение из командной строки.
grep(process.stdin, process.stdout, pattern) // Вызвать асинхронную
                                             // функцию grep().
    .catch(err => {                  // Обработать асинхронные исключения.
        console.error(err);
        process.exit();
    });
```

16.5.3. Запись в потоки и обработка противодавления

Асинхронная функция `grep()` в предыдущем примере кода продемонстрировала, как применять поток `Readable` в качестве асинхронного итератора, а также то, что вы можете записывать данные в поток `Writable`, просто передавая их методу `write()`. Метод `write()` принимает в своем первом аргументе буфер или строку. (Объектные потоки ожидают другие виды объектов, но в настоящей главе они не рассматриваются.) Если вы передаете буфер, тогда

байты этого буфера будут записываться напрямую. Если вы передаете строку, то перед записью она будет закодирована в буфер байтов. Потоки `Writable` имеют стандартную кодировку, которая используется в случае передачи строки методу `write()` как единственного аргумента. Стандартной кодировкой обычно будет `"utf8"`, но вы можете установить ее явно, вызвав метод `setDefaultEncoding()` для потока `Writable`. Или же при передаче методу `write()` строки в первом аргументе вы можете передать во втором аргументе имя кодировки.

В необязательном третьем аргументе метод `write()` принимает функцию обратного вызова, которая будет вызываться, когда данные фактически записаны и больше не находятся во внутреннем буфере потока `Writable`. (Обратный вызов также может быть вызван при возникновении ошибки, но это не гарантируется. Чтобы обнаруживать ошибки, вы должны зарегистрировать для потока `Writable` обработчик событий `"error"`.)

Метод `write()` имеет очень важное возвращаемое значение. Когда вы вызываете метод `write()` для потока, он будет всегда принимать и буферизировать порцию переданных ему данных. Затем он возвращает `true`, если внутренний буфер пока еще не полон. Если же буфер уже полон или переполнен, тогда `write()` возвращает `false`. Такое возвращаемое значение носит рекомендательный характер, и вы можете проигнорировать его — потоки `Writable` будут при необходимости расширять свои внутренние буферы, если вы продолжите вызывать `write()`. Но не забывайте, что причина применения потокового API-интерфейса в первую очередь заключается в том, чтобы избежать затрат, связанных с хранением в памяти крупных объемов данных.

Возвращаемое значение `false` метода `write()` является формой *противодавления* (или обратного давления; backpressure), т.е. сообщением от потока о том, что вы записали данные быстрее, чем они могут быть обработаны.

Надлежащая реакция на противодавление такого рода предусматривает прекращение вызова `write()` до тех пор, пока поток не выпустит событие `"drain"` (опустошен), сигнализирующее о том, что в буфере снова есть место. Например, вот функция, которая записывает в поток и затем вызывает обратный вызов, когда в поток можно записывать новые данные:

```
function write(stream, chunk, callback) {
    // Записать указанную порцию в указанный поток.
    let hasMoreRoom = stream.write(chunk);

    // Проверить возвращаемое значение метода write():
    if (hasMoreRoom) {          // Если он возвратил true, тогда
        setImmediate(callback); // асинхронно вызвать обратный вызов.
    } else {                    // Если он возвратил false, тогда
        stream.once("drain", callback);  // вызвать обратный вызов
                                         // для события "drain".
    }
}
```

Тот факт, что иногда допустимо вызывать метод `write()` много раз подряд, а временами приходится ожидать какого-то события между операциями записи,

порождает неуклюжие алгоритмы. Это одна из причин, из-за которых использовать метод `pipe()` настолько привлекательно: когда вы применяете `pipe()`, среда Node обрабатывает противодавление автоматически.

Если вы используете в своей программе `await` и `async` и обращаетесь с потоками `Readable` как с асинхронными итераторами, тогда будет несложно реализовать основанную на `Promise` версию служебной функции `write()`, чтобы должным образом обрабатывать противодавление. В ранее показанной асинхронной функции `grep()` мы не обрабатывали противодавление. В приведенной далее асинхронной функции `copy()` демонстрируется, как правильно делать такую обработку. Обратите внимание, что функция `copy()` только копирует порции из потока источника в поток назначения, а вызов `copy(source, destination)` во многом похож на вызов `source.pipe(destination)`:

```
// Эта функция записывает указанную порцию в указанный поток
// и возвращает объект Promise, который будет удовлетворен,
// когда запись снова допускается. Поскольку она возвращает
// объект Promise, ее можно использовать с await.
function write(stream, chunk) {
    // Записать указанную порцию в указанный поток.
    let hasMoreRoom = stream.write(chunk);

    if (hasMoreRoom) {                    // Если буфер не полон, тогда
        return Promise.resolve(null);     // возвратить уже разрешенный
                                          // объект Promise.
    } else {
        return new Promise(resolve => {   // В противном случае
                                          // возвратить объект Promise,
            stream.once("drain", resolve); // который разрешается
                                          // в событие "drain".
        });
    }
}

// Копировать данные из потока источника в поток назначения,
// обрабатывая противодавление из потока назначения.
// Это во многом похоже на вызов source.pipe(destination).
async function copy(source, destination) {
    // Настроить обработчик ошибок для потока назначения на случай
    // непредвиденного закрытия стандартного вывода.
    destination.on("error", err => process.exit());

    // Использовать цикл for/await для асинхронного чтения
    // порций из потока ввода.
    for await (let chunk of source) {
        // Записать порцию и ожидать, пока не появится больше места
        // в буфере.
        await write(destination, chunk);
    }
}

// Копировать стандартный ввод в стандартный вывод.
copy(process.stdin, process.stdout);
```

Прежде чем завершить обсуждение записи в потоки, важно еще раз отметить, что отсутствие реакции на противодавление может привести к тому, что ваша программа станет потреблять больше памяти, чем должна, когда внутренний буфер потока `Writable` переполняется и продолжает свой рост. Если вы реализуете сетевой сервер, то это может быть удаленно эксплуатируемой проблемой безопасности. Предположим, что вы создали HTTP-сервер, который доставляет файлы по сети, но не применяли `pipe()` и не уделили время обработке противодавления из метода `write()`. Злоумышленник может написать HTTP-клиент, который инициирует запросы для крупных файлов (таких как изображения), но никогда в действительности не читает тело запроса. Поскольку клиент не читает данные по сети, а сервер не реагирует на противодавление, буферы на сервере начинают переполняться. При достаточном количестве одновременных подключений от злоумышленника ситуация может превратиться в атаку типа отказа от обслуживания, которая замедлит сервер или даже приведет к его аварийному отказу.

16.5.4. Чтение потоков с помощью событий

Потоки `Readable` в Node поддерживают два режима, каждый из которых имеет собственный API-интерфейс для чтения. Если вы не можете использовать каналы или асинхронную итерацию в своей программе, тогда для работы с потоками вам нужно будет выбрать один из двух API-интерфейсов на основе событий. Важно применять либо один, либо другой, но не смешивать оба API-интерфейса.

Режим извлечения данных

В *режиме извлечения данных* (flowing mode), когда поступают читабельные данные, они немедленно выпускаются в форме события `"data"`. Чтобы читать поток в таком режиме, просто зарегистрируйте обработчик для событий `"data"` и поток будет проталкивать вам порции данных (буферы или строки), как только они становятся доступными. Обратите внимание, что вызывать метод `read()` в режиме извлечения данных не нужно: вам необходимо только обрабатывать события `"data"`. Имейте в виду, что вновь созданные потоки не запускаются в режиме извлечения данных. Регистрация обработчика событий `"data"` переключает поток в режим извлечения данных. Это означает, что поток не выпускает события `"data"` до тех пор, пока вы не зарегистрируете первый обработчик событий `"data"`.

Если вы используете режим извлечения для чтения данных из потока `Readable`, их обработки и записи в поток `Writable`, тогда вам может потребоваться обработка противодавления из потока `Writable`. Если метод `write()` возвращает `false`, указывая на то, что буфер записи полон, то вы можете вызвать метод `pause()` потока `Readable`, чтобы временно прекратить выпуск событий `"data"`. Получив затем событие `"drain"` из потока `Writable`, вы можете вызвать метод `resume()` потока `Readable`, чтобы снова началось протекание событий `"data"`.

Поток в режиме извлечения данных выпускает событие "end", когда достигнут конец потока. Событие "end" указывает на то, что события "data" больше не будут выпускаться. И подобно всем потокам событие "error" выпускается в случае возникновения ошибки.

В начале раздела о потоках приводилась непотоковая функция copyFile() и было обещано реализовать ее улучшенную версию. В следующем коде показана потоковая функция copyFile(), которая применяет API-интерфейс режима извлечения данных и обрабатывает противодавление. То же самое было бы легче реализовать посредством вызова pipe(), но код служит полезной демонстрацией использования множества обработчиков событий, которые применяются для координирования протекания данных из одного потока в другой.

```
const fs = require("fs");

// Потоковая функция копирования файлов, использующая режим
// извлечения данных. Копирует содержимое указанного файла источника
// в указанный файл назначения.
// В случае успеха вызывает обратный вызов с аргументом null.
// В случае ошибки вызывает обратный вызов с объектом Error.
function copyFile(sourceFilename, destinationFilename, callback) {
    let input = fs.createReadStream(sourceFilename);
    let output = fs.createWriteStream(destinationFilename);

    input.on("data", (chunk) => {      //Когда мы получим новые данные,
        let hasRoom = output.write(chunk); //записать их в поток вывода
        if (!hasRoom) {          // Если поток вывода полон,
            input.pause();       // тогда приостановить поток ввода.
        }
    });
    input.on("end", () => {  // Когда мы достигли конца потока ввода,
        output.end();        // сообщить потоку вывода об окончании
    });
    input.on("error", err => { // Если мы получили ошибку в потоке ввода,
        callback(err);         // тогда вызвать обратный вызов с ошибкой
        process.exit();        // и завершить работу.
    });

    output.on("drain", () => {   // Когда поток вывода больше не полон,
        input.resume();          // возобновить выпуск событий "data"
                                 // потоком ввода.
    });
    output.on("error", err => {  // Если мы получили ошибку в потоке
        callback(err);           // вывода, тогда вызвать обратный
        process.exit();          // вызов с ошибкой и завершить работу.
    });
    output.on("finish", () => {  // Когда поток вывода полностью записан,
        callback(null);      // вызвать обратный вызов со значением null.
    });
}

// Простая утилита командной строки для копирования файлов.
let from = process.argv[2], to = process.argv[3];
console.log(`Copying file ${from} to ${to}...`);
```

```
copyFile(from, to, err => {
   if (err) {
      console.error(err);      // Возникла ошибка.
   } else {
      console.log("done.");    // Готово.
   }
});
```

Режим ожидания

Другой режим потоков `Readable` называется *режимом ожидания* (paused mode). Именно в нем запускаются потоки. Если вы не регистрировали обработчик событий `"data"` и не вызывали метод `pipe()`, тогда поток `Readable` остается в режиме ожидания, в котором он не проталкивает вам данные в форме событий `"data"`. Взамен вы извлекаете данные из потока, явно вызывая его метод `read()`. Вызов не является блокирующим, и если в потоке нет данных, доступных для чтения, то он возвращает `null`. Так как синхронный API-интерфейс для ожидания данных отсутствует, API-интерфейс режима ожидания тоже основан на событиях. Поток `Readable` в режиме ожидания выпускает события `"readable"`, когда данные становятся доступными для чтения из потока. В ответ ваш код должен вызвать метод `read()`, чтобы прочитать эти данные. Вы обязаны организовать цикл, многократно вызывая метод `read()` до тех пор, пока он не возвратит `null`. Для того чтобы в будущем инициировалось новое событие `"readable"`, необходимо подобным образом полностью опустошить буфер потока. Если вы прекратите вызывать `read()`, несмотря на наличие доступных для чтения данных, то не получите еще одного события `"readable"` и ваша программа, скорее всего, зависнет.

Потоки в режиме ожидания выпускают события `"end"` и `"error"` в точности как потоки в режиме извлечения данных. Если вы разрабатываете программу, которая читает данные из потока `Readable` и записывает их в поток `Writable`, тогда режим ожидания может оказаться не лучшим вариантом. Для надлежащей обработки противодавления вы хотите читать, только когда поток ввода читабелен и поток вывода не заполнен. В режиме ожидания это означает чтение и запись до тех пор, пока метод `read()` не возвратит `null` или метод `write()` не возвратит `false`, и затем снова чтение или запись согласно событиям `"readable"` или `"drain"`. Описанный подход не особенно элегантен и вы можете обнаружить, что в таком случае легче использовать режим извлечения данных (или каналы).

В приведенном далее коде показано, как можно вычислить хеш-значение SHA256 для содержимого указанного файла. Здесь с применением потока `Readable` в режиме ожидания читается содержимое файла порциями, и каждая порция передается объекту, который вычисляет хеш-значение. (Обратите внимание, что в Node 12 и последующих версиях такую функцию проще написать с использованием цикла `for/await`.)

```
const fs = require("fs");
const crypto = require("crypto");
```

```
// Вычисляет хеш-значение SHA256 для содержимого указанного файла
// и передает это хеш-значение (как строку) указанной функции
// обратного вызова с первым аргументом-ошибкой.
function sha256(filename, callback) {
    let input = fs.createReadStream(filename); // Поток данных.
    let hasher = crypto.createHash("sha256");  // Для вычисления хеша.
    input.on("readable", () => {    // Когда есть данные, доступные
                                    // для чтения.
        let chunk;
        while(chunk = input.read()) { // Читать порцию, и если она не null,
            hasher.update(chunk);     // тогда передать ее в hasher;
        }                         // продолжать цикл, пока данные читабельны.
    });
    input.on("end", () => {               // В конце потока
        let hash = hasher.digest("hex");  // вычислить хеш
        callback(null, hash);         // и передать его обратному вызову.
    });
    input.on("error", callback);          // В случае ошибки вызвать
                                          // обратный вызов.
}
// Простая утилита командной строки для вычисления хеша файла.
sha256(process.argv[2], (err, hash) => { // Передать имя файла
                                         // из командной строки.
    if (err) {                           // Если возникла ошибка,
        console.error(err.toString());   // вывести сообщение о ней.
    } else {                             // В противном случае
        console.log(hash);               // вывести строку с хешем.
    }
});
```

16.6. Информация о процессе, центральном процессоре и операционной системе

Глобальный объект `Process` имеет несколько полезных свойств и функций, которые обычно связаны с состоянием выполняющегося в текущий момент процесса Node. Полные сведения ищите в документации Node, а ниже представлен ряд свойств и функций, о которых вам следует знать:

```
process.argv          // Массив аргументов командной строки.
process.arch          // Архитектура ЦП: например, "x64".
process.cwd()         // Возвращает текущий рабочий каталог.
process.chdir()       // Устанавливает текущий рабочий каталог.
process.cpuUsage()    // Сообщает коэффициент загрузки ЦП.
process.env           // Объект с переменными среды.
process.execPath      // Абсолютный путь к исполняемому файлу Node
                      // в файловой системе.
process.exit()        // Прекращает выполнение программы.
process.exitCode      // Целочисленный код, который будет сообщен
                      // при завершении программы.
```

```
process.getuid()            // Возвращает идентификатор пользователя
                            // Unix для текущего пользователя.
process.hrtime.bigint()     // Возвращает наносекундную отметку
                            // времени с высоким разрешением.
process.kill()              // Посылает сигнал другому процессу.
process.memoryUsage()       // Возвращает объект со сведениями
                            // об использовании памяти.
process.nextTick()          // Как и setImmediate(), вызывает функцию
                            // в ближайшее время.
process.pid                 // Идентификатор текущего процесса.
process.ppid                // Идентификатор родительского процесса.
process.platform            // Операционная система: например,
                            // "linux", "darwin" или "win32".
process.resourceUsage()     // Возвращает объект со сведениями
                            // об использовании ресурсов.
process.setuid()            // Устанавливает текущего пользователя
                            // по идентификатору или по имени.
process.title               // Имя процесса, которое появляется
                            // в листингах команды ps.
process.umask()             // Устанавливает или возвращает стандартные
                            // разрешения для новых файлов.
process.uptime()            // Возвращает время безотказной
                            // работы Node в секундах.
process.version             // Строка с версией Node.
process.versions            // Строки с версиями библиотек,
                            // от которых зависит Node.
```

Модуль os (который в отличие от process должен быть явно загружен с помощью require()) обеспечивает доступ к похожей низкоуровневой информации о компьютере и ОС, под управлением которой выполняется Node. Возможно, ни одно из таких средств вам никогда не понадобится, но стоит знать о том, что Node делает их доступными:

```
const os = require("os");
os.arch()     // Возвращает архитектуру ЦП: например, "x64" или "arm".
os.constants        // Полезные константы, такие
                    // как os.constants.signals.SIGINT.
os.cpus()           // Данные о ядрах ЦП системы, включая время загрузки.
os.endianness()     // Собственный порядок байтов ЦП: "BE" или "LE".
os.EOL              // Собственный разделитель строк ОС: "\n" или "\r\n".
os.freemem()        // Возвращает объем свободной оперативной памяти
                    // в байтах.
os.getPriority()    // Возвращает запланированный ОС приоритет процесса.
os.homedir()        // Возвращает домашний каталог текущего пользователя.
os.hostname()       // Возвращает имя хоста компьютера.
os.loadavg()        // Возвращает средние показатели загрузки за 1, 5
                    // и 15 минут.
os.networkInterfaces() // Возвращает сведения о доступных
                       // сетевых подключениях.
os.platform()          // Возвращает ОС: например, "linux", "darwin"
                       // или "win32".
```

```
os.release()          // Возвращает номер версии ОС.
os.setPriority()      // Пытается установить запланированный
                      // приоритет для процесса.
os.tmpdir()           // Возвращает стандартный временный каталог.
os.totalmem()         // Возвращает общий объем оперативной памяти
                      // в байтах.
os.type()             // Возвращает ОС: например, "Linux", "Darwin"
                      // или "Windows_NT".
os.uptime()           // Возвращает время безотказной работы
                      // системы в секундах.
os.userInfo()         // Возвращает идентификатор,имя,домашний каталог
                      // и командную оболочку текущего пользователя.
```

16.7. Работа с файлами

Модуль `fs` в Node — это всеобъемлющий API-интерфейс для работы с файлами и каталогами. Его дополняет модуль `path`, в котором определены служебные функции для работы с именами файлов и каталогов. Модуль `fs` содержит несколько высокоуровневых функций для легкого чтения, записи и копирования файлов. Но большинство функций в модуле являются низкоуровневыми привязками JavaScript к системным вызовам Unix (и их эквивалентам в Windows). Если вы ранее работали с низкоуровневыми вызовами, касающимися файловой системы (в C или других языках), тогда API-интерфейс `fs` будет выглядеть для вас знакомым. Если же нет, то некоторые части API-интерфейса `fs` могут показаться не особо понятными. Скажем, функция для удаления файла называется `unlink()`.

В модуле `fs` определен крупный API-интерфейс главным образом из-за наличия множества разновидностей каждой фундаментальной операции. Как обсуждалось в начале главы, большинство функций вроде `fs.readFile()` являются неблокирующими, основанными на обратных вызовах и асинхронными. Однако обычно каждая такая функция имеет синхронный блокирующий вариант, такой как `fs.readFileSync()`. В Node 10 и последующих версиях многие из этих функций также имеют основанные на `Promise` асинхронные варианты наподобие `fs.promises.readFile()`. Большинство функций `fs` принимают в своем первом аргументе строку, указывающую путь (имя файла плюс необязательные имена каталогов) к файлу, с которым нужно работать. Но несколько функций также поддерживают варианты, принимающие в первом аргументе целочисленный "файловый дескриптор" вместо пути. Такие варианты имеют имена, которые начинаются с буквы "f". Например, `fs.truncate()` усекает файл, заданный путем, а `fs.ftruncate()` усекает файл, заданный файловым дескриптором. Существует основанная на `Promise` версия `fs.promises.truncate()`, которая ожидает путь, и еще одна версия на основе `Promise`, которая реализована в виде метода объекта `FileHandle`. (Класс `FileHandle` является эквивалентом файлового дескриптора в API-интерфейсе, основанном на `Promise`.) Наконец, в модуле `fs` есть ряд функций, которые имеют варианты с именами, начинающимися с буквы "l". Они похожи на базовую функцию, но не следуют по символическим ссылкам в файловой системе, а оперируют прямо на самих символических ссылках.

16.7.1. Пути, файловые дескрипторы и объекты FileHandle

Чтобы применять модуль fs для работы с файлом, сначала необходимо иметь возможность именования желаемого файла. Файлы чаще всего указываются с помощью *пути*, который означает имя самого файла плюс иерархия каталогов, где файл находится. Если путь *абсолютный*, то указываются все каталоги вплоть до корня файловой системы. В противном случае путь является *относительным* и имеет смысл только по отношению к какому-то другому пути, обычно *текущему рабочему каталогу*. Работа с путями может оказаться немного сложной, т.к. из-за того, что разные ОС используют отличающиеся символы для отделения имен каталогов, при объединении путей эти разделительные символы легко случайно удвоить, а также потому, что сегменты пути к родительскому каталогу ../ требуют специальной обработки. В данной ситуации помогает модуль path и другие важные средства Node:

```
// Некоторые важные пути.
process.cwd()  // Абсолютный путь к текущему рабочему каталогу.
__filename     // Абсолютный путь к файлу, который хранит текущий код.
__dirname      // Абсолютный путь к каталогу,
               // в котором находится __filename.
os.homedir()   // Домашний каталог пользователя.

const path = require("path");

path.sep       // Либо "/", либо "\" в зависимости от ОС.

// Модуль path содержит простые функции разбора.
let p = "src/pkg/test.js";      // Пример пути.
path.basename(p)                // => "test.js"
path.extname(p)                 // => ".js"
path.dirname(p)                 // => "src/pkg"
path.basename(path.dirname(p)) // => "pkg"
path.dirname(path.dirname(p))  // => "src"

// Функция normalize() очищает пути:
path.normalize("a/b/c/../d/")  // =>"a/b/d/":обрабатывает сегменты ../
path.normalize("a/./b")        // => "a/b": убирает сегменты ./
path.normalize("//a//b//")     // => "/a/b/": удаляет дублированные /

// Функция join() объединяет сегменты пути,
// добавляя разделители, и нормализует их.
path.join("src", "pkg", "t.js") // => "src/pkg/t.js"

// Функция resolve() принимает один или большее количество сегментов
// пути и возвращает абсолютный путь. Она начинает с последнего
// аргумента и останавливается, когда построит абсолютный путь
// или разрешается в process.cwd().
path.resolve()                  // => process.cwd()
path.resolve("t.js")            // => path.join(process.cwd(), "t.js")
path.resolve("/tmp", "t.js")    // => "/tmp/t.js"
path.resolve("/a", "/b", "t.js") // => "/b/t.js"
```

Обратите внимание, что `path.normalize()` — просто функция манипулирования строками, которая не имеет доступа к фактической файловой системе. Функции `fs.realpath()` и `fs.realpathSync()` выполняют канонические преобразования в контексте файловой системы: они распознают символические ссылки и интерпретируют относительные пути относительно текущего рабочего каталога.

В предшествующих примерах мы предполагали, что код выполнялся в среде ОС Unix и `path.sep` было `"/"`. Если вы хотите работать с путями в стиле Unix даже в системе Windows, тогда применяйте `path.posix` вместо `path`. И наоборот, если вы желаете работать с путями Windows даже в системе Unix, то `path.win32`. `path.posix` и `path.win32` определяют такие же свойства и функции, как сам `path`.

Некоторые функции `fs`, обсуждаемые в последующих подразделах, ожидают *файловый дескриптор*, а не имя файла. Файловые дескрипторы представляют собой целые числа, используемые как ссылки на "открытые" файлы. Вы получаете дескриптор для заданного имени с помощью вызова функции `fs.open()` (или `fs.openSync()`). Процессам разрешено иметь ограниченное количество одновременно открытых файлов, поэтому важно, чтобы вы вызывали `fs.close()` для своих файловых дескрипторов, когда закончите с ними работу. Открывать файлы необходимо, если вы хотите применять функции самого низкого уровня `fs.read()` и `fs.write()`, которые позволяют перескакивать внутри файла, читая и записывая его биты в разное время. В модуле `fs` существуют и другие функции, использующие файловые дескрипторы, но все они имеют версии на основе имен файлов, а применять функции, основанные на файловых дескрипторах, имеет смысл лишь в случае, если вы так или иначе собираетесь открывать файл для чтения или записи.

Наконец, в API-интерфейсе на основе `Promise`, определенном в `fs.promises`, эквивалентом `fs.open()` является функция `fs.promises.open()`, которая возвращает объект `Promise`, разрешаемый в объект `FileHandle`. Такой объект `FileHandle` служит той же цели, что и файловый дескриптор. Тем не менее, если только вам не нужно использовать методы самого низкого уровня `read()` и `write()` объекта `FileHandle`, то на самом деле нет никаких причин для его создания. И если вы все же создаете объект `FileHandle`, то не должны забывать о вызове его метода `close()` по завершении работы с ним.

16.7.2. Чтение из файлов

Среда Node позволяет читать содержимое файла все целиком, через поток или с помощью низкоуровневого API-интерфейса.

Если ваши файлы небольшие или расход памяти и производительность не являются наивысшим приоритетом, тогда часто легче прочитать все содержимое файла посредством единственного вызова. Вы можете делать это синхронно, с применением обратного вызова или с использованием объекта `Promise`. По умолчанию вы получите байты файла в виде буфера, но если указана кодировка, то взамен вы получите декодированную строку.

```
const fs = require("fs");
let buffer = fs.readFileSync("test.data");   // Синхронно,
                                             // возвращается буфер.
let text = fs.readFileSync("data.csv","utf8"); // Синхронно,
                                             // возвращается строка.

// Читать байты файла асинхронно.
fs.readFile("test.data", (err, buffer) => {
    if (err) {
        // Обработать ошибку.
    } else {
        // Байты файла находятся в буфере.
    }
});

// Асинхронное чтение на основе Promise.
fs.promises
    .readFile("data.csv", "utf8")
    .then(processFileText)
    .catch(handleReadError);

// Или применить API-интерфейс Promise с await внутри
// асинхронной функции.
async function processText(filename, encoding="utf8") {
    let text = await fs.promises.readFile(filename, encoding);
    // ...обработать текст...
}
```

Если у вас есть возможность обрабатывать содержимое файла последовательно и отсутствует необходимость, чтобы все содержимое файла находилось в памяти, тогда чтение файла через поток может оказаться самым эффективным подходом. Потоки были подробно рассмотрены ранее: ниже показано, как с применением потока и метода `pipe()` записать содержимое файла в стандартный вывод:

```
function printFile(filename, encoding="utf8") {
    fs.createReadStream(filename, encoding).pipe(process.stdout);
}
```

Наконец, если вас интересует низкоуровневый контроль над тем, какие именно байты вы читаете из файла и когда, то можете открыть файл для получения файлового дескриптора и затем использовать функцию `fs.read()`, `fs.readSync()` или `fs.promises.read()`, чтобы прочитать указанное количество байтов из заданного местоположения источника в указанный буфер, расположенный в заданной позиции назначения:

```
const fs = require("fs");

// Чтение указанной порции файла данных.
fs.open("data", (err, fd) => {
    if (err) {
        // Сообщить об ошибке.
        return;
    }
```

```
    try {
        // Прочитать байты с 20 по 420 во вновь выделенный буфер.
        fs.read(fd, Buffer.alloc(400), 0, 400, 20, (err, n, b) => {
            // err - ошибка, если есть.
            // n - количество фактически прочитанных байтов.
            // b - буфер, в который были прочитаны байты.
        });
    }
    finally {          // Использовать конструкцию finally, чтобы всегда
        fs.close(fd); // закрывать открытый файловый дескриптор.
    }
});
```

Основанный на обратных вызовах API-интерфейс `read()` неудобен в применении, когда вам необходимо прочитать более одной порции данных из файла. Если вы можете использовать синхронный API-интерфейс (или API-интерфейс на основе `Promise` с `await`), то читать множество порций из файла будет легко:

```
const fs = require("fs");

function readData(filename) {
    let fd = fs.openSync(filename);
    try {
        // Прочитать заголовок файла.
        let header = Buffer.alloc(12);          // Буфер из 12 байтов.
        fs.readSync(fd, header, 0, 12, 0);

        // Проверить магическое число файла.
        let magic = header.readInt32LE(0);
        if (magic !== 0xDADAFEED) {
            throw new Error("File is of wrong type");
                        // Файл имеет неправильный тип
        }

        // Получить из заголовка смещение и длину данных.
        let offset = header.readInt32LE(4);
        let length = header.readInt32LE(8);

        // Прочитать эти байты из файла.
        let data = Buffer.alloc(length);
        fs.readSync(fd, data, 0, length, offset);
        return data;
    } finally {

        // Всегда закрывать файл, даже если
        // было сгенерировано исключение.
        fs.closeSync(fd);
    }
}
```

16.7.3. Запись в файлы

Запись в файлы в Node очень похожа на чтение из файлов, но с ней связано несколько дополнительных деталей, о которых следует знать. Одна из деталей заключается в том, что вы создаете новый файл, просто выполняя запись в указанное имя файла, который пока не существует.

Как и в случае чтения, среда Node предлагает три основных способа записи в файлы. При наличии полного содержимого файла в строке или в буфере вы можете записать его посредством одного вызова функции `fs.writeFile()` (основана на обратных вызовах), `fs.writeFileSync()` (синхронная) или `fs.promises.writeFile()` (основана на `Promise`):

```
fs.writeFileSync(path.resolve(__dirname, "settings.json"),
                 JSON.stringify(settings));
```

Если данные, записываемые в файл, находятся в строке, но вы хотите применять кодировку, отличающуюся от `"utf8"`, тогда передайте желаемую кодировку в необязательном третьем аргументе.

Связанные функции `fs.appendFile()`, `fs.appendFileSync()` и `fs.promises.appendFile()` похожи, но когда указанный файл уже существует, они добавляют данные в конец файла, а не переписывают имеющееся содержимое. Если данные, которые вы хотите записать в файл, не расположены в одной порции или не находятся одновременно в памяти, тогда использование потока `Writable` будет хорошим подходом в предположении, что вы планируете записывать данные с начала до конца, не делая пропусков в файле:

```
const fs = require("fs");
let output = fs.createWriteStream("numbers.txt");
for(let i = 0; i < 100; i++) {
   output.write(`${i}\n`);
}
output.end();
```

Наконец, если вы хотите записывать в файл данные в виде множества порций и располагать возможностью контроля точной позиции внутри файла, куда записывается каждая порция, то можете открыть файл с помощью `fs.open()`, `fs.openSync()` или `fs.promises.open()` и применять результирующий файловый дескриптор с функцией `fs.write()` или `fs.writeSync()`. Упомянутые функции имеют разные формы для строк и буферов. Вариант для строки принимает файловый дескриптор, строку и позицию в файле, в которую должна быть записана строка (с кодировкой в необязательном четвертом аргументе). Вариант для буфера принимает файловый дескриптор, буфер, смещение и длину, указывающие порцию данных внутри буфера, а также позицию в файле, в которую должны быть записаны байты этой порции. И если у вас есть массив объектов `Buffer`, который вы хотите записать, то можете сделать это с помощью единственного вызова `fs.writev()` или `fs.writevSync()`. Для записи буферов и строк существуют подобные низкоуровневые функции, которые используют функцию `fs.promises.open()` и возвращаемый ею объект `FileHandle`.

Строки режима доступа к файлу

Вы уже видели функции `fs.open()` и `fs.openSync()` ранее при работе с низкоуровневым API-интерфейсом для чтения файлов. Тогда было достаточно просто передать имя файла функции открытия. Однако если вы хотите записывать в файл, то должны также передать второй строковый аргумент, который указывает, как планируется применять файловый дескриптор.

Вот некоторые из доступных строк флагов.

`"w"`

: Открыть файл для записи.

`"w+"`

: Открыть файл для записи и чтения.

`"wx"`

: Создать новый файл и открыть его для записи; терпит неудачу, если указанный файл уже существует.

`"wx+"`

: Создать новый файл и открыть его для записи и чтения; терпит неудачу, если указанный файл уже существует.

`"a"`

: Открыть файл для добавления; существующее содержимое не перезаписывается.

`"a+"`

: Открыть файл для добавления, но также разрешить чтение.

Если вы не передадите одну из перечисленных выше строк флагов функции `fs.open()` или `fs.openSync()`, то она будет использовать стандартный флаг `"r"`, делая файловый дескриптор допускающим только чтение. Обратите внимание, что эти флаги полезно передавать также другим функциям для записи в файлы:

```
// Записать в файл за один вызов, но добавить к тому,
// что в нем уже есть.
// Работает подобно fs.appendFileSync().
fs.writeFileSync("messages.log", "hello", { flag: "a" });

// Открыть поток записи, но генерировать ошибку,
// если файл уже существует.
// Мы не хотим случайно что-то перезаписать!
// Обратите внимание, что параметром выше был flag, а здесь - flags.
fs.createWriteStream("messages.log", { flags: "wx" });
```

Вы можете усекать файл с помощью `fs.truncate()`, `fs.truncateSync()` или `fs.promises.truncate()`. Упомянутые функции принимают путь в первом аргументе и длину во втором и модифицируют файл так, чтобы он имел указанную длину. Если длина опущена, тогда в качестве нее применяется ноль и файл становится пустым. Несмотря на имена этих функций, их можно также использовать для расширения файла: если указывается длина, превышающая текущий размер файла, то файл расширяется нулевыми байтами до нового размера. Если вы уже открыли файл, который хотите модифицировать, тогда можете применять `ftruncate()` или `ftruncateSync()` с файловым дескриптором или объектом `FileHandle`.

Описанные здесь разнообразные функции для записи в файлы производят возврат или вызывают свой обратный вызов либо разрешают свой объект `Promise`, когда данные были "записаны" в том смысле, что среда Node передала их ОС. Но это не обязательно означает, что данные уже фактически записаны в постоянное хранилище: по крайней мере, часть ваших данных может все еще быть буферизованной где-то ОС или драйвером устройства в ожидании записи на диск. Если вы вызываете `fs.writeSync()` для асинхронной записи каких-то данных в файл и немедленно после возврата управления из функции происходит сбой электропитания, то вы по-прежнему можете потерять данные. Если вы хотите принудительно выгрузить данные на диск, чтобы точно знать, что они благополучно сохранились, тогда используйте `fs.fsync()` или `fs.fsyncSync()`. Указанные функции работают только с файловыми дескрипторами: версий на основе путей не предусмотрено.

16.7.4. Файловые операции

В предыдущем обсуждении классов потоков Node приводились два примера функции `copyFile()`. Они не были практичными утилитами, которые вы действительно применяли бы, т.к. в модуле `fs` определена собственная функция `fs.copyFile()` (а также `fs.copyFileSync()` и `fs.promises.copyFile()`).

Упомянутые функции принимают в своих первых двух аргументах имя исходного файла и имя копии, которые можно указывать в виде строк, URL или объектов `Buffer`. Необязательный третий аргумент — это целое число, биты которого указывают флаги, управляющие деталями операции копирования. И для основанной на обратных вызовах функции `fs.copyFile()` в последнем аргументе передается функция обратного вызова, которая будет вызываться, когда копирование завершится, или будет вызываться с аргументом ошибки, если что-то пошло не так. Вот несколько примеров:

```
// Базовое синхронное копирование файла.
fs.copyFileSync("ch15.txt", "ch15.bak");

// Аргумент COPYFILE_EXCL обеспечивает копирование, только если
// новый файл не существует. Он предотвращает перезаписывание
// существующих файлов.
fs.copyFile("ch15.txt", "ch16.txt", fs.constants.COPYFILE_EXCL, err => {
    // Этот обратный вызов будет вызван, когда копирование завершено.
    // В случае ошибки err будет не null.
});
```

```
// Следующий код демонстрирует применение версии функции copyFile()
// на основе Promise. Два файла объединяются с помощью побитовой
// операции ИЛИ |. Флаги означают, что существующие файлы не будут
// перезаписываться, и если файловая система поддерживает это,
// то копия будет клоном копирования при записи исходного файла,
// т.е. дополнительное пространство для хранения не потребуется
// до тех пор, пока оригинал или копия не будут модифицированы.
fs.promises.copyFile("Important data",
        `Important data ${new Date().toISOString()}`
        fs.constants.COPYFILE_EXCL | fs.constants.COPYFILE_FICLONE)
   .then(() => {
      console.log("Backup complete");
                  // Копирование завершено
   });
   .catch(err => {
      console.error("Backup failed", err);
                   // Копирование потерпело неудачу
   });
```

Функция `fs.rename()` (наряду с обычными синхронным и основанным на `Promise` вариантами) перемещает и/или переименовывает файл. Вызывайте ее с текущим путем к файлу и желательным новым путем к файлу. Аргументов с флагами нет, но версия на основе обратных вызовов принимает в третьем аргументе обратный вызов:

```
fs.renameSync("ch15.bak", "backups/ch15.bak");
```

Обратите внимание, что флага, который не допустил бы перезаписывания существующего файла при переименовании, не предусмотрено. Также имейте в виду, что файлы могут быть переименованы только в рамках файловой системы.

Функции `fs.link()` и `fs.symlink()` и их варианты имеют такие же сигнатуры, как `fs.rename()`, и ведут себя немного похоже на `fs.copyFile()` за исключением того, что они создают соответственно жесткие и символические ссылки, а не копии.

В заключение отметим, что `fs.unlink()`, `fs.unlinkSync()` и `fs.promises.unlink()` являются функциями Node для удаления файла. (Не совсем понятные имена функций унаследованы из Unix, где удаление файла по существу противоположно созданию жесткой ссылки на него.) Вызывайте указанные функции с путем к удаляемому файлу в виде строки, буфера или URL и передавайте обратный вызов, если используете версию на основе обратных вызовов:

```
fs.unlinkSync("backups/ch15.bak");
```

16.7.5. Метаданные файлов

Функции `fs.stat()`, `fs.statSync()` и `fs.promises.stat()` позволяют получать метаданные для указанного файла или каталога. Например:

```
const fs = require("fs");
let stats = fs.statSync("book/ch15.md");
stats.isFile()          // => true: это обыкновенный файл.
stats.isDirectory()     // => false: это не каталог.
stats.size              // Размер файла в байтах.
stats.atime             // Время доступа: дата и время,
                        // когда было последнее чтение.
stats.mtime             // Время изменения: дата и время,
                        // когда была последняя запись.
stats.uid               // Идентификатор пользователя владельца файла.
stats.gid               // Идентификатор группы владельца файла.
stats.mode.toString(8) // Разрешения файла в форме строки
                        // с восьмеричным числом.
```

Возвращенный объект `Stats` содержит другие, менее понятные свойства и методы, но в коде были продемонстрированы те, с которыми вы вероятнее всего будете работать.

Функция `fs.lstat()` и ее варианты работают так же, как `fs.stat()`, за исключением того, что если указанный файл является символической ссылкой, то Node возвратит метаданные для самой ссылки, а не проследует по ссылке.

Если вы открыли файл, чтобы создать файловый дескриптор или объект `FileHandle`, то можете применять функцию `fs.fstat()` или ее варианты для получения метаданных, относящихся к открытому файлу, без необходимости в повторном указании имени файла.

Помимо запрашивания метаданных посредством функции `fs.stat()` и всех ее вариантов существуют также функции для изменения метаданных.

Функции `fs.chmod()`, `fs.lchmod()` и `fs.fchmod()` (вместе с синхронными и основанными на `Promise` версиями) устанавливают "режим" или разрешения файла либо каталога. Значения режима представляют собой целые числа, в которых каждый бит имеет особый смысл, а потому их легче воспринимать в восьмеричной форме записи. Скажем, чтобы сделать файл допускающим только чтение для его владельца и недоступным для всех остальных, используйте `0o400`:

```
fs.chmodSync("ch15.md", 0o400);  // Чтобы не удалить его случайно!
```

Функции `fs.chown()`, `fs.lchown()` и `fs.fchown()` (наряду с синхронными и основанными на `Promise` версиями) устанавливают владельца и группу (в виде идентификаторов) для файла или каталога. (Они важны, потому что взаимодействуют с разрешениями файла, установленными функцией `fs.chmod()`.)

Наконец, вы можете установить время доступа и время изменения файла или каталога с помощью функций `fs.utimes()` и `fs.futimes()` и их вариантов.

16.7.6. Работа с каталогами

Для создания нового каталога в Node применяйте функцию `fs.mkdir()`, `fs.mkdirSync()` или `fs.promises.mkdir()`. В первом аргументе передается путь создаваемого каталога. Необязательный второй аргумент может быть целым числом, которое указывает режим (биты разрешений) для нового каталога.

Либо же вы можете передать объект с необязательными свойствами mode и recursive. Если recursive равно true, то функция создаст любые каталоги в пути, которые пока еще не существуют:

```
// Обеспечить существование dist/ и dist/lib/.
fs.mkdirSync("dist/lib", { recursive: true });
```

Функция fs.mkdtemp() и ее варианты принимают предоставляемый вами префикс пути, добавляют к нему ряд случайных символов (это важно для безопасности), создают каталог с таким именем и возвращают вам (или передают обратному вызову) путь каталога.

Чтобы удалить каталог, используйте функцию fs.rmdir() или один из ее вариантов. Обратите внимание, что каталоги должны быть пустыми, прежде чем они могут быть удалены:

```
// Создать случайный временный каталог и получить его путь,
// затем удалить его, когда мы закончили с ним работать.
let tempDirPath;
try {
    tempDirPath = fs.mkdtempSync(path.join(os.tmpdir(), "d"));
    // Делать что-то с каталогом.
} finally {
    // Удалить временный каталог, когда мы закончили с ним работать.
    fs.rmdirSync(tempDirPath);
}
```

Модуль fs предлагает два отдельных API-интерфейса для вывода содержимого каталога. Функции fs.readdir(), fs.readdirSync() и fs.promises.readdir() читают сразу весь каталог и предоставляют массив строк или массив объектов Dirent, который указывает имена и типы (файл или каталог) каждого элемента. Имена файлов, возвращенные указанными функциями, будут просто локальными именами файла, не полными путями. Ниже приведены примеры:

```
let tempFiles = fs.readdirSync("/tmp");  // Возвращает массив строк.

// Использовать API-интерфейс на основе Promise для получения
// массива Dirent и затем вывести пути подкаталогов.
fs.promises.readdir("/tmp", {withFileTypes: true})
    .then(entries => {
        entries.filter(entry => entry.isDirectory())
            .map(entry => entry.name)
            .forEach(name => console.log(path.join("/tmp/", name)));
    })
    .catch(console.error);
```

Если вы ожидаете, что вам придется выводить содержимое каталогов, которые могут иметь тысячи элементов, тогда вы можете отдать предпочтение потоковому подходу, реализованному в функции fs.opendir() и ее вариантах. Эти функции возвращают объект Dir, представляющий указанный каталог. Вы можете применять метод read() или readSync() объекта Dir для чтения по одному объекту Dirent за раз. Если вы передаете методу read() функцию об-

ратного вызова, то он будет вызывать ее. И если вы опустите аргумент обратного вызова, то метод read() возвратит объект Promise. Когда элементов в каталоге больше нет, вместо объекта Dirent вы получите null.

Объекты Dir легче всего использовать как асинхронные итераторы в цикле for/await. Например, вот функция, которая применяет потоковый API-интерфейс для вывода элементов каталога, вызывает для каждого элемента stat() и выводит имена и размеры файлов и каталогов:

```
const fs = require("fs");
const path = require("path");

async function listDirectory(dirpath) {
    let dir = await fs.promises.opendir(dirpath);
    for await (let entry of dir) {
        let name = entry.name;
        if (entry.isDirectory()) {
          name += "/"; //Добавлять к подкаталогам хвостовую косую черту
        }
        let stats = await fs.promises.stat(path.join(dirpath, name));
        let size = stats.size;
        console.log(String(size).padStart(10), name);
    }
}
```

16.8. Клиенты и серверы HTTP

Модули http, https и http2 в Node являются полнофункциональными, но относительно низкоуровневыми реализациями протоколов HTTP. В них определены всеобъемлющие API-интерфейсы для построения клиентов и серверов HTTP. Поскольку эти API-интерфейсы относительно низкоуровневые, раскрыть все их возможности в одной главе не удастся. В показанных далее примерах будет продемонстрировано создание базовых клиентов и серверов.

Простейший способ выполнения базового HTTP-запроса GET предусматривает использование http.get() или https.get(). В первом аргументе этих функций передается URL для извлечения. (В случае URL вида http:// вы должны применять модуль http, а в случае URL вида https:// — модуль https.) Во втором аргументе указывается обратный вызов, который будет вызываться с объектом IncomingMessage, как только начнет поступать ответ сервера. Когда обратный вызов вызывается, состояние и заголовки HTTP доступны, но тело может быть еще не готово. Объект IncomingMessage представляет собой поток Readable, и для чтения из него тела ответа вы можете использовать методики, рассмотренные ранее в главе.

В функции getJSON(), описанной в конце раздела 13.2.6, была задействована функция http.get() как часть демонстрации применения конструктора Promise(). Теперь, когда вы больше знаете о потоках и программной модели Node, стоит вернуться к указанному примеру и посмотреть, как в нем использовалась функция http.get().

Функции `http.get()` и `https.get()` являются несколько упрощенными вариантами более общих функций `http.request()` и `https.request()`. В следующей функции `postJSON()` иллюстрируется применение `https.request()` для выдачи HTTPS-запроса POST, который включает тело запроса в формате JSON. Как и функция `getJSON()` из главы 13, она ожидает ответа в формате JSON и возвращает объект `Promise`, который удовлетворяется разобранной версией данного ответа:

```
const https = require("https");
/*
 * Преобразует объект тела в строку JSON, затем передает ее с помощью
 * HTTPS-запроса POST указанной конечной точке API-интерфейса
 * в указанном хосте. Когда поступает ответ, разбирает тело ответа как
 * JSON и разрешает возвращенный объект Promise разобранным значением.
 */
function postJSON(host, endpoint, body, port, username, password) {
    // Немедленно возвращает объект Promise и затем вызывает resolve
    //или reject, когда HTTPS-запрос проходит успешно или терпит неудачу
    return new Promise((resolve, reject) => {
        // Преобразовать люънет тела в строку.
        let bodyText = JSON.stringify(body);

        // Сконфигурировать HTTPS-запрос.
        let requestOptions = {
            method: "POST",    // Или "GET", "PUT", "DELETE" и т.д.
            host: host,        // Хост, к которому нужно подключиться.
            path: endpoint,    // Путь URL.
            headers: {         // Заголовки HTTP для запроса.
                "Content-Type": "application/json",
                "Content-Length": Buffer.byteLength(bodyText)
            }
        };
        if (port) {                     // Если порт указан, тогда
          requestOptions.port = port;// использовать его для запроса.
        }
        // Если учетные данные указаны, тогда добавить
        // заголовок авторизации.
        if (username && password) {
            requestOptions.auth = `${username}:${password}`;
        }

        // Создать запрос на основе объекта конфигурации.
        let request = https.request(requestOptions);

        // Записать тело запроса POST и закончить запрос.
        request.write(bodyText);
        request.end();

        // Отклонить в случае возникновения ошибок с запросом
        // (таких как отсутствие сетевого подключения).
        request.on("error", e => reject(e));
```

```
        // Обработать ответ, когда он начнет поступать.
        request.on("response", response => {
            if (response.statusCode !== 200) {
                reject(new Error(`HTTP status ${response.statusCode}`));
                // В этом случае нас не интересует тело ответа, но мы
                // не хотим, чтобы оно оставалось где-то в буфере,
                // а потому переводим поток в режим извлечения данных,
                // не регистрируя обработчик для событий "data",
                // так что тело отбрасывается.
                response.resume();
                return;
            }
            // Нам нужен текст, не байты. Мы предполагаем, что
            // текст будет в формате JSON, но не проверяем
            // заголовок Content-Type.
            response.setEncoding("utf8");
            // Среда Node не имеет потокового анализатора JSON,
            // поэтому мы читаем целое тело ответа в строку.
            let body = "";
            response.on("data", chunk => { body += chunk; });
            // А теперь обработать ответ, когда он завершит
            // свое поступление.
            response.on("end", () => { // Когда ответ закончен,
                try {                  // попытаться разобрать его как JSON
                    resolve(JSON.parse(body));  // и разрешить результат.
                } catch(e) {           // Или, если что-то пошло не так,
                    reject(e);         // то отклонить с ошибкой.
                }
            });
        });
    });
}
```

Помимо выдачи запросов HTTP и HTTPS модули `http` и `https` также позволяют создавать серверы, которые реагируют на такие запросы. Ниже описан базовый подход.

- Создать объект `Server`.
- Вызвать его метод `listen()`, чтобы начать прослушивание запросов на указанном порту.
- Зарегистрировать обработчик для событий `"request"`, использовать его для чтения запроса клиента (в частности свойства `request.url`) и записать свой ответ.

Следующий код создает простой HTTP-сервер, обслуживающий статические файлы в локальной файловой системе, и также реализует отладочную конечную точку, которая реагирует на запрос клиента, возвращая обратно этот запрос.

```
// Простой статический HTTP-сервер, который обслуживает файлы
// из указанного каталога. Также реализует специальную конечную
// точку /test/mirror, которая возвращает обратно входящий запрос,
// что удобно при отладке клиентов.
const http = require("http");   // Используйте "https" при
                                // наличии сертификата.
const url = require("url");     // Для разбора URL.
const path = require("path");   // Для манипулирования путями
                                // в файловой системе.
const fs = require("fs");       // Для чтения файлов.

// Обслуживает файлы из указанного корневого каталога
// через HTTP-сервер, который прослушивает указанный порт.
function serve(rootDirectory, port) {
    let server = new http.Server();  // Создать новый HTTP-сервер.
    server.listen(port);             // Прослушивать указанный порт.
    console.log("Listening on port", port);

    // Когда поступает запрос, обработать его с помощью этой функции.
    server.on("request", (request, response) => {
        // Получить порцию пути URL запроса, игнорируя
        // любые параметры, добавленные к запросу.
        let endpoint = url.parse(request.url).pathname;

        // Если запрос был к "/test/mirror", тогда послать запрос
        // обратно в том же виде. Полезно, когда нужно видеть
        // заголовки и тело запроса.
        if (endpoint === "/test/mirror") {
            // Установить заголовок ответа.
            response.setHeader("Content-Type", "text/plain;
                               charset=UTF-8");

            // Указать код состояния ответа.
            response.writeHead(200);  // 200 OK

            // Начать тело ответа с запроса.
            response.write(`${request.method} ${request.url} HTTP/${
                           request.httpVersion
                          }\r\n`);

            // Вывести заголовки запроса.
            let headers = request.rawHeaders;
            for(let i = 0; i < headers.length; i += 2) {
                response.write(`${headers[i]}: ${headers[i+1]}\r\n`);
            }

            // Закончить заголовки дополнительной пустой строкой.
            response.write("\r\n");

            // Теперь необходимо скопировать тело запроса в тело ответа.
            // Поскольку они оба являются потоками, можно использовать
            // канал.
            request.pipe(response);
        }
```

```
        // В противном случае обслужить файл из локального каталога.
        else {
            // Отобразить конечную точку на файл в локальной
            // файловой системе.
            let filename = endpoint.substring(1);  // Убрать ведущий
                                                   // символ "/".
            // Не разрешать "../" в пути, потому что это приведет к бреш
            // в системе безопасности, позволяющей обслуживать что угодно
            // вне корневого каталога.
            filename = filename.replace(/\.\.\//g, "");
            // Преобразовать относительное имя файла в абсолютное.
            filename = path.resolve(rootDirectory, filename);

            // Выяснить тип содержимого файла на основе его расширения.
            let type;
            switch(path.extname(filename)) {
                case ".html":
                case ".htm": type = "text/html"; break;
                case ".js":  type = "text/javascript"; break;
                case ".css": type = "text/css"; break;
                case ".png": type = "image/png"; break;
                case ".txt": type = "text/plain"; break;
                default:     type = "application/octet-stream"; break;
            }

            let stream = fs.createReadStream(filename);
            stream.once("readable", () => {
                // Если поток становится читабельным, тогда установить
                // заголовок Content-Type и состояние 200 OK. Затем
                // организовать канал между читаемым потоком и потоком
                // ответа. Канал автоматически вызовет response.end(),
                // когда поток закончится.
                response.setHeader("Content-Type", type);
                response.writeHead(200);
                stream.pipe(response);
            });

            stream.on("error", (err) => {
                // Если при попытке открыть поток мы получаем ошибку,
                // то файл, вероятно, не существует или не допускает
                // чтение. Послать простой текстовый ответ 404 Not Found
                // с сообщением об ошибке.
                response.setHeader("Content-Type", "text/plain;
                                   charset=UTF-8");
                response.writeHead(404);
                response.end(err.message);
            });
        }
    });
}

// При обращении из командной строки вызывать функцию serve().
serve(process.argv[2] || "/tmp", parseInt(process.argv[3]) || 8000);
```

Встроенные модули Node — это все, что необходимо для реализации простых серверов HTTP и HTTPS. Тем не менее, имейте в виду, что производственные серверы обычно не строятся непосредственно на встроенных модулях Node. Взамен большинство сложных серверов реализуются с применением внешних библиотек, таких как фреймворк Express, которые обеспечивают "промежуточное программное обеспечение" и другие утилиты более высокого уровня, ожидаемые разработчиками стороны сервера.

16.9. Сетевые серверы и клиенты, не использующие HTTP

Веб-серверы и клиенты настолько распространены, что легко забыть о возможности создания клиентов и серверов, которые не используют HTTP. Несмотря на то что Node имеет репутацию хорошей среды для написания веб-серверов, она также обладает полной поддержкой для реализации других типов сетевых серверов и клиентов.

Если вам удобно работать с потоками, тогда взаимодействовать с сетью будет просто, потому что сетевые сокеты являются разновидностью потока `Duplex`. В модуле `net` определены классы `Server` и `Socket`. Чтобы создать сервер, вызовите функцию `net.createServer()` и затем метод `listen()` результирующего объекта для сообщения серверу порта, который он должен прослушивать на предмет подключений. Когда клиент подключается к указанному порту, объект `Server` генерирует событие `"connection"`, а значением, передаваемым прослушивателю событий, будет объект `Socket`. Объект `Socket` — это поток `Duplex`, который вы можете применять для чтения данных из клиента и записи данных в клиент. Вызовите метод `end()` объекта `Socket`, чтобы отключиться.

Реализовать клиент даже проще: передайте функции `net.createConnection()` номер порта и имя хоста, чтобы создать сокет для взаимодействия с сервером, который запущен на указанном хосте и прослушивает указанный порт. Затем используйте результирующий сокет для чтения и записи данных на сервер.

В следующем коде показано, как написать сервер с помощью модуля `net`. При подключении клиента сервер сообщает шутку типа "тук-тук":

```
// Сервер TCP, который вырабатывает интерактивные шутки
// типа "тук-тук" на порту 6789.
// (Почему шестерка боится семерки? Потому что семерка съела девятку!)
const net = require("net");
const readline = require("readline");

// Создать объект Server и начать прослушивание подключений.
let server = net.createServer();
server.listen(6789, () => console.log("Delivering laughs on port 6789"));
                                    // Вырабатывание шуток на порту 6789

// Когда клиент подключается, сообщить ему шутку типа "тук-тук".
server.on("connection", socket => {
   tellJoke(socket)
      .then(() => socket.end()) //Когда шутка закончена,закрыть сокет
```

```
        .catch((err) => {
            console.error(err);   // Вывести сообщения о любых
                                  // случившихся ошибках,
            socket.end();         // но все равно закрыть сокет!
        });
});
// Это все известные нам шутки.
const jokes = {
    "Boo": "Don't cry...it's only a joke!",
    "Lettuce": "Let us in! It's freezing out here!",
    "A little old lady": "Wow, I didn't know you could yodel!"
    // "Парень": "Не плачь... это всего лишь шутка!",
    // "Деньги": "Впусти нас! Здесь холодно!",
    // "Маленькая старушка": "Здорово, я и не знала,
    // что ты умеешь петь йодлем!"
};
// Интерактивно исполнить шутку типа "тук-тук" по этому сокету
// без блокирования.
async function tellJoke(socket) {
    // Выбрать случайным образом одну из шуток.
    let randomElement = a => a[Math.floor(Math.random() * a.length
    let who = randomElement(Object.keys(jokes));
    let punchline = jokes[who];

    // Использовать модуль readline для чтения пользовательского
    // ввода по одной строке за раз.
    let lineReader = readline.createInterface({
        input: socket,
        output: socket,
        prompt: ">> "
    });

    // Служебная функция для вывода клиенту строки текста
    // и затем (по умолчанию) отображения приглашения на ввод.
    function output(text, prompt=true) {
        socket.write(`${text}\r\n`);
        if (prompt) lineReader.prompt();
    }

    // Шутки типа "тук-тук" имеют структуру вида "оклик и ответ".
    // Мы ожидаем от пользователя разного ввода на разных стадиях
    // и предпринимаем разные действия, когда получаем ввод
    // на разных стадиях.
    let stage = 0;

    // Начать шутку типа "тук-тук" в традиционной манере.
    output("Knock knock!");

    // Асинхронно читать строки из клиента до тех пор,
    // пока шутка не закончится.
    for await (let inputLine of lineReader) {
        if (stage === 0) {
            if (inputLine.toLowerCase() === "who's there?") {
```

```
                // Если пользователь дает правильный ответ на стадии 0,
                // тогда сообщить ему первую часть шутки и перейти
                // на стадию 1.
                output(who);
                stage = 1;
            } else {
                // В противном случае научить пользователя,
                // как обращаться с шутками типа "тук-тук".
                output('Please type "Who\'s there?".');
                      // Пожалуйста, введите Who's there?.
            }
        } else if (stage === 1) {
            if (inputLine.toLowerCase()===`${who.toLowerCase()} who?`) {
                // Если ответ пользователя корректен на стадии 1, тогда
                // выработать концовку и выполнить возврат,
                // т.к. шутка закончена.
                output(`${punchline}`, false);
                return;
            } else {
                // Вынудить пользователя подыгрывать.
                output(`Please type "${who} who?".`);
                      // Пожалуйста, введите <имя> who?.
            }
        }
    }
}
```

Простые серверы на основе текста вроде этого обычно не нуждаются в специальном клиенте. Если в вашей системе установлена утилита nc ("netcat"), то вы можете применять ее для взаимодействия с реализованным сервером:

```
$ nc localhost 6789
Knock knock!
Тук-тук!
>> Who's there?
>> Кто там?
A little old lady
Маленькая старушка
>> A little old lady who?
>> Маленькая старушка кто?
Wow, I didn't know you could yodel!
Здорово, я и не знала, что ты умеешь петь йодлем!
```

С другой стороны, написать в Node специальный клиент для сервера выдачи шуток очень легко. Мы просто подключаемся к серверу, затем соединяем каналом вывод сервера и stdout, после чего соединяем каналом stdin и ввод сервера:

```
// Подключиться к порту шуток (6789) на сервере с именем,
// указанным в командной строке.
let socket = require("net").createConnection(6789, process.argv[2]);
```

```
socket.pipe(process.stdout);                    // Канал для данных между
                                                // сокетом и stdout.
process.stdin.pipe(socket); // Канал для данных между stdin и сокетом.
socket.on("close", () => process.exit()); // Завершить, когда сокет
                                                // закрывается.
```

В дополнение к поддержке серверов на основе TCP модуль `net` среды Node также поддерживает межпроцессное взаимодействие через "сокеты домена Unix", которые идентифицируются путем в файловой системе, а не номером порта. Сокеты такого рода в главе не рассматриваются, но детали можно найти в документации Node. Другие средства Node, на раскрытие которых здесь не хватило места, включают модуль `dgram` для основанных на UDP клиентов и серверов и модуль `tls`, соответствующий `net` как `https` соответствует `http`. Классы `tls.Server` и `tls.TLSSocket` позволяют создавать серверы TCP (подобные серверу выдачи шуток типа "тук-тук"), которые используют подключения, зашифрованные с помощью SSL, как в серверах HTTPS.

16.10. Работа с дочерними процессами

Помимо реализации серверов с высокой степенью параллелизма Node также хорошо подходит для написания сценариев, выполняющих другие программы. В модуле `child_process` среды Node определено несколько функций для запуска других программ как дочерних процессов. В последующих подразделах описаны некоторые из таких функций, начиная с простейших и заканчивая более сложными.

16.10.1. `execSync()` и `execFileSync()`

Запустить еще одну программу проще всего посредством функции `child_process.execSync()`. Она принимает в первом аргументе команду, подлежащую запуску. Функция `execSync()` создает дочерний процесс, запускает в нем командную оболочку и применяет оболочку для выполнения переданной вами команды. Затем она блокируется до тех пор, пока команда (и оболочка) не закончит работу. Если команда завершается с ошибкой, тогда `execSync()` генерирует исключение. В противном случае `execSync()` возвращает вывод, который команда записала в поток `stdout`. По умолчанию таким возвращаемым значением является буфер, но вы можете указать в необязательном втором аргументе кодировку, чтобы взамен получить строку. Если команда записывает любой вывод в поток `stderr`, то он просто передается в поток `stderr` родительского процесса.

Таким образом, например, если вы пишете сценарий и не беспокоитесь о производительности, то можете использовать `child_process.execSync()` для получения списка содержимого каталога с помощью знакомой команды оболочки Unix, а не функции `fs.readdirSync()`:

```
const child_process = require("child_process");
let listing =
    child_process.execSync("ls -l web/*.html", {encoding: "utf8"});
```

Тот факт, что `execSync()` вызывает полную командную оболочку Unix, означает возможность включения в передаваемую строку множества разделенных точками с запятой команд и получения преимуществ от таких средств оболочки, как групповые символы в именах файлов, конвейеры и перенаправление вывода. Это также означает, что вы обязаны проявлять осторожность и никогда не передавать функции `execSync()` команду, любая порция которой является пользовательским вводом или поступает из аналогичного ненадежного источника. Сложный синтаксис команд оболочки можно легко изменить, чтобы злоумышленник сумел запустить произвольный код.

Если вам не нужны средства командной оболочки, тогда вы можете избежать накладных расходов, связанных с запуском оболочки, применяя функцию `child_process.execFileSync()`, которая выполняет программу напрямую без вызова оболочки. Но поскольку оболочка не задействована, она не может проводить разбор командной строки, так что вам придется передавать исполняемый файл в первом аргументе и массив аргументов командной строки во втором:

```
let listing = child_process.execFileSync("ls", ["-l", "web/"],
                                         {encoding: "utf8"});
```

Параметры дочерних процессов

Функция `execSync()` и многие другие функции `child_process` принимают необязательный второй или третий аргумент, в котором указываются дополнительные детали относительно того, как должен запускаться дочерний процесс. Свойство `encoding` этого объекта использовалось ранее для указания о том, что мы хотим доставки вывода команды в виде строки, а не буфера. Ниже перечислены остальные важные свойства, которые вы можете указывать (следует отметить, что не все параметры доступны всем функциям дочернего процесса).

- `cwd` указывает рабочий каталог для дочернего процесса. Если вы опустите его, тогда дочерний процесс наследует значение `process.cwd()`.
- `env` указывает переменные среды, к которым будет иметь доступ дочерний процесс. По умолчанию дочерние процессы просто наследуют `process.env`, но при желании вы можете задать другой объект.
- `input` указывает строку или буфер входных данных, которые должны применяться как стандартный ввод для дочернего процесса. Этот параметр доступен только синхронным функциям, не возвращающим объект `ChildProcess`.
- `maxBuffer` указывает максимальное количество байтов вывода, которые будут собраны функциями `exec`. (Данный параметр не применим к функциям `spawn()` и `fork()`, которые используют потоки.) Если дочерний процесс производит больше вывода, чем задано в `maxBuffer`, то он будет уничтожен и завершится с ошибкой.
- `shell` указывает путь к исполняющему файлу командной оболочки или `true`. Для функций дочернего процесса, которые обычно выполняют ко-

манду оболочки, этот параметр позволяет указать, какую оболочку применять. Для функций, которые обычно не используют оболочку, данный параметр позволяет указать, что оболочка должна применяться (за счет установки свойства в `true`), или указать, какая именно оболочка должна использоваться.

- `timeout` задает максимальное количество миллисекунд, в течение которых дочернему процессу должно быть разрешено выполняться. Если дочерний процесс не завершится до того, как истечет это время, то он будет уничтожен и закончится с ошибкой. (Данный параметр применим к функциям exec, но не к `spawn()` или `fork()`.)
- `uid` указывает идентификатор пользователя (число), от имени которого должна запускаться программа. Если родительский процесс выполняется от имени привилегированной учетной записи, тогда этот параметр можно использовать для запуска дочернего процесса с сокращенными привилегиями.

16.10.2. `exec()` и `execFile()`

Функции `execSync()` и `execFileSync()`, как указывают их имена, являются синхронными: они блокируются и не возвращают управление до тех пор, пока не завершится дочерний процесс. Применение данных функций во многом похоже на ввод команд Unix в окне терминала: они позволяют запускать последовательность команд за раз. Но если вы пишете программу, которой необходимо выполнить несколько задач, и задачи никак не зависят друг от друга, тогда у вас может возникнуть желание распараллелить их и запускать множество команд одновременно. Достичь такой цели можно с помощью асинхронных функций `child_process.exec()` и `child_process.execFile()`.

Функции `exec()` и `execFile()` похожи на свои синхронные варианты за исключением того, что они производят немедленный возврат с объектом `ChildProcess`, который представляет запущенный дочерний процесс, и принимают в своем последнем аргументе обратный вызов с первым аргументом-ошибкой. Обратный вызов вызывается, когда дочерний процесс завершает работу, и фактически вызывается с тремя аргументами. Первый аргумент — ошибка, если она возникла; в случае нормального завершения процесса он будет равен `null`. Второй аргумент — накопленный вывод, который был послан стандартному потоку вывода дочернего процесса. Третий аргумент — вывод, отправленный стандартному потоку ошибок дочернего процесса.

Объект `ChildProcess`, возвращаемый `exec()` и `execFile()`, позволяет прекращать работу дочернего процесса и записывать в него данные (которые затем можно читать из его стандартного ввода). Объект `ChildProcess` будет описан при обсуждении функции `child_process.spawn()`.

Если вы планируете выполнять множество дочерних процессов одновременно, тогда может быть легче использовать версию функции `exec()`, возвращающую объект `Promise`, который в случае завершения дочернего процесса без ошибок разрешается в объект со свойствами `stdout` и `stderr`. Ниже приведе-

на функция, которая принимает массив команд оболочки и возвращает объект Promise, разрешаемый в результат всех переданных команд:

```
const child_process = require("child_process");
const util = require("util");
const execP = util.promisify(child_process.exec);

function parallelExec(commands) {
    // Использовать массив команд для создания массива объектов Promise.
    let promises =
        commands.map(command => execP(command, {encoding: "utf8"}));
    // Возвратить объект Promise, который будет удовлетворен с массивом
    // удовлетворенных значений каждого индивидуального объекта Promise.
    // (Вместо возвращения объектов со свойствами stdout и stderr
    // мы просто возвращаем значение stdout.)
    return Promise.all(promises)
        .then(outputs => outputs.map(out => out.stdout));
}

module.exports = parallelExec;
```

16.10.3. spawn()

Описанные до сих пор разнообразные функции exec — синхронные и асинхронные — спроектированы для применения с дочерними процессами, которые выполняются быстро и не производят много вывода. Даже синхронные функции exec() и execFile() не являются потоковыми: они возвращают вывод процесса в единственном пакете только после завершения работы процесса.

Функция child_process.spawn() позволяет осуществлять потоковый доступ к выводу дочернего процесса, пока процесс все еще выполняется. Она также дает возможность записывать данные в дочерний процесс (который будет видеть записанные данные как ввод в своем стандартном потоке ввода): это означает, что с дочерним процессом можно динамически взаимодействовать, посылая ему ввод на основе генерируемого им вывода.

По умолчанию функция spawn() не использует командную оболочку, так что вы должны вызывать ее подобно execFile() с исполняемым файлом, подлежащим запуску, и отдельным массивом аргументов командной строки, которые будут ему передаваться. Как и execFile(), функция spawn() возвращает объект ChildProcess, но не принимает аргумент с обратным вызовом. Вместо применения функции обратного вызова вы прослушиваете события для объекта ChildProcess и его потоков данных.

Возвращаемый функцией spawn() объект ChildProcess представляет собой генератор событий. Вы можете прослушивать событие "exit", чтобы получать уведомление о завершении работы дочернего процесса. Объект ChildProcess также имеет три потоковых свойства. stdout и stderr — это потоки Readable: когда дочерний процесс записывает в свои стандартные потоки вывода и ошибок, такой вывод становится читабельным через потоки данных ChildProcess. Обратите здесь внимание на инверсию имен. В дочернем процессе "стандартным потоком вывода" является поток вывода Writable, но

в родительском процессе свойство stdout объекта ChildProcess представляет собой поток ввода Readable.

Аналогично свойство stdin объекта ChildProcess будет потоком Writeable: все, что вы записываете в него, становится доступным дочернему процессу через его стандартный ввод.

В объекте ChildProcess также определено свойство pid, которое хранит идентификатор дочернего процесса. Вдобавок в нем определен метод kill(), позволяющий прекращать работу дочернего процесса.

16.10.4. fork()

child_process.fork() является специализированной функцией для запуска модуля кода JavaScript в дочернем процессе Node. Функция fork() ожидает такие же аргументы, как и spawn(), но первый аргумент должен указывать путь к файлу кода JavaScript, а не к двоичному исполняемому файлу.

Дочерний процесс, созданный функцией fork(), может взаимодействовать с родительским процессом через свои стандартные потоки ввода и вывода, как было описано в предыдущем подразделе, посвященном spawn(). Но вдобавок функция fork() обеспечивает еще один более легкий канал связи между родительским и дочерним процессами.

Когда вы создаете дочерний процесс с помощью fork(), то можете использовать метод send() возвращенного объекта ChildProcess для отправки копии какого-то объекта дочернему процессу. И вы можете прослушивать событие "message" в объекте ChildProcess, чтобы получать сообщения от дочернего процесса. В коде, выполняющемся в дочернем процессе, можно применять process.send() для отправки сообщения родительскому процессу и прослушивать события "message" в process, чтобы получать сообщения от родительского процесса. Например, ниже показан код, где с использованием функции fork() создается дочерний процесс, затем дочернему процессу посылается сообщение и ожидается ответ:

```
const child_process = require("child_process");

// Запустить новый процесс Node, выполняющий код из child.js
// в нашем каталоге.
let child = child_process.fork(`${__dirname}/child.js`);

// Послать сообщение дочернему процессу.
child.send({x: 4, y: 3});

// Вывести ответ дочернего процесса, когда он прибудет.
child.on("message", message => {
    console.log(message.hypotenuse);  // Это должно вывести 5.
    // Поскольку мы послали только одно сообщение, то ожидаем только
    // один ответ. После его получения мы вызываем disconnect(), чтобы
    // закрыть подключение между родительским и дочерним процессами.
    // Это дает возможность процессам чисто завершить работу.
    child.disconnect();
});
```

А вот код, который выполняется в дочернем процессе:

```
// Ожидать сообщения от родительского процесса.
process.on("message", message => {
    // Когда сообщение получено, провести расчет
    // и отправить результат родительскому процессу.
    process.send({hypotenuse: Math.hypot(message.x, message.y)});
});
```

Запуск дочерних процессов — дорогостоящая операция, и дочерний процесс должен был бы выполнять на порядки больше вычислений, чтобы появился смысл в подобном применении `fork()` и межпроцессного взаимодействия. Если вы пишете программу, которая обязана быстро реагировать на входящие события и также выполнять отнимающие много времени вычисления, тогда рассмотрите возможность использования отдельного дочернего процесса для выполнения вычислений, чтобы они не блокировали цикл обработки событий и не снижали отзывчивость родительского процесса. (Хотя в таком сценарии поток — см. раздел 16.11 — может оказаться лучшим вариантом, чем дочерний процесс.)

Первый аргумент `send()` будет сериализироваться посредством `JSON.stringify()` и десериализироваться в дочернем процессе с помощью `JSON.parse()`, поэтому вы должны включать только значения, поддерживаемые форматом JSON. Однако в методе `send()` предусмотрен специальный второй аргумент, который позволяет передавать дочернему процессу объекты `Socket` и `Server` (из модуля `net`). Сетевые серверы обычно ограничены скоростью ввода-вывода, а не вычислений, но если вы написали сервер, который должен выполнять больше вычислений, чем способен обеспечить единственный ЦП, и запускаете этот сервер на машине с множеством ЦП, тогда можете применять `fork()` для создания множества дочерних процессов, обрабатывающих запросы. В родительском процессе вы будете прослушивать события `"connection"` в объекте `Server`, затем извлекать объект `Socket` из события `"connection"` и посылать его с использованием специального второго аргумента `send()` одному из дочерних процессов на обработку. (Обратите внимание, что это непривлекательное решение необычного сценария. Вместо написания сервера, который ответвляет дочерние процессы, вероятно проще оставить сервер однопоточным, а для обработки нагрузки развернуть в производственной среде несколько экземпляров сервера.)

16.11. Потоки воркеров

Как объяснялось в начале главы, модель параллелизма Node является однопоточной и основанной на событиях. Но в Node 10 и последующих версиях стало возможным подлинное многопоточное программирование с применением API-интерфейса, который близко отображается на API-интерфейс веб-воркеров, определенный в веб-браузерах (см. раздел 15.13). Многопоточное программирование имеет заслуженную репутацию сложного занятия. Сложность почти полностью связана с необходимостью тщательной синхронизации доступа потоков к разделяемой памяти. Но потоки JavaScript (в Node и в браузерах) по умолча-

нию не разделяют память, поэтому опасности и трудности использования потоков к "воркерам" в JavaScript не относятся.

Вместо применения разделяемой памяти потоки воркеров JavaScript взаимодействуют путем передачи сообщений. Главный поток может посылать сообщение потоку воркера, вызывая метод `postMessage()` объекта `Worker`, который представляет этот поток. Поток воркера способен получать сообщения от своего родительского потока, прослушивая события `"message"`. И воркеры с помощью собственной версии `postMessage()` могут посылать сообщения главному потоку, которые он может получать посредством собственного обработчика событий `"message"`. Приведенный далее пример кода прояснит, как все работает.

Есть три причины, по которым вы можете решить использовать потоки воркеров в приложении Node.

- Если вашему приложению действительно нужно выполнять больше вычислений, чем способно обработать одно ядро ЦП, тогда потоки позволят распределить работу среди множества ядер, что в наши дни стало обычным явлением. Если вы занимаетесь научными расчетами, машинным обучением или обработкой графических данных в Node, то можете применять потоки просто ради того, чтобы увеличить вычислительную мощность для решения имеющейся задачи.
- Даже когда ваше приложение не задействует на полную мощь один ЦП, у вас по-прежнему может возникать желание использовать потоки для обеспечения отзывчивости главного потока. Рассмотрим сервер, который обрабатывает крупные, но относительно нечастые запросы. Предположим, что он получает только один запрос в секунду, но ему необходимо потратить около половины секунды на вычисления (блокирующие ЦП) для обработки каждого запроса. В среднем сервер будет простаивать 50% времени. Но когда поступают два запроса, между которыми проходит несколько миллисекунд, сервер даже не сможет начать ответ на второй запрос, пока не завершатся вычисления для первого запроса. Если же сервер применяет для выполнения вычислений поток воркера, тогда он может немедленно начать ответ на оба запроса и обеспечить лучший отклик клиентам сервера. При условии, что сервер имеет более одного ядра ЦП, он может также вычислять тела обоих запросов параллельно, но даже при наличии единственного ядра использование воркеров все равно улучшает отзывчивость.
- В целом воркеры позволяют превращать блокирующие синхронные операции в неблокирующие асинхронные. Если вы пишете программу, зависящую от унаследованного кода, который неизбежно является синхронным, тогда вы можете применять воркеры, когда нужно вызывать такой унаследованный код.

Потоки воркеров не настолько тяжеловесны как дочерние процессы, но они и не легковесны. Обычно нет смысла создавать воркер, если только вы не располагаете значительным объемом работ для него. И, вообще говоря, если ваша программа не связана с ЦП и не имеет проблем с отзывчивостью, то вам вряд ли понадобятся потоки воркеров.

16.11.1. Создание воркеров и передача сообщений

Воркеры определены в модуле Node по имени `worker_threads`. В этом подразделе мы будем ссылаться на него с помощью идентификатора `threads`:

```
const threads = require("worker_threads");
```

Данный модуль определяет класс `Worker` для представления потока воркера, и новый поток можно создать с помощью конструктора `threads.Worker()`. Приведенный далее код использует конструктор `threads.Worker()` для создания воркера и показывает, как передавать сообщения из главного потока в воркер и из воркера в главный поток. Также в коде демонстрируется трюк, позволяющий размещать код главного потока и потока воркера в том же самом файле[2].

```
const threads = require("worker_threads");

// Модуль worker_threads экспортирует булевское свойство isMainThread.
// Это свойство равно true, когда Node выполняет главный поток,
// и false, когда Node выполняет поток воркера. Мы можем использовать
// данный факт для реализации главного потока и потока воркера
// в одном и том же файле.
if (threads.isMainThread) {
    // Если мы находимся в главном потоке, тогда мы всего лишь
    // экспортируем функцию. Вместо выполнения интенсивной в плане
    // вычислений задачи в главном потоке эта функция передает задачу
    // воркеру и возвращает объект Promise, который будет разрешен,
    // когда воркер закончит работу.
    module.exports = function reticulateSplines(splines) {
        return new Promise((resolve,reject) => {
            // Создать воркер, который загружает и запускает этот же
            // файл кода.
            // Обратите внимание на использование специальной
            // переменной __filename.
            let reticulator = new threads.Worker(__filename);

            // Передать воркеру копию массива splines.
            reticulator.postMessage(splines);

            // Затем разрешить или отклонить объект Promise, когда
            // мы получаем из воркера сообщение или ошибку.
            reticulator.on("message", resolve);
            reticulator.on("error", reject);
        });
    };
} else {
    // Если мы попали сюда, то это значит, что мы находимся в воркере,
    // а потому мы регистрируем обработчик для получения сообщений
    // из главного потока.
```

[2] Часто бывает проще и яснее помещать код потока воркера в отдельный файл. Но этот трюк с двумя потоками, запускающими разные части одного файла, поразил мое воображение, когда я впервые столкнулся с ним применительно к системному вызову `fork()` в Unix. И я полагаю, что такой прием стоит продемонстрировать просто из-за его необычной элегантности.

```
    // Данный воркер предназначен для приема только одного сообщения,
    // поэтому мы регистрируем обработчик событий с помощью once(),
    // а не on(). В таком случае воркер естественным образом
    // завершится, когда закончит свою работу.
    threads.parentPort.once("message", splines => {
        // Когда мы получили массив splines из главного потока,
        // пройти в цикле по его элементам и покрыть их сетчатым узором
        // (reticulate).
        for(let spline of splines) {
            // В качестве примера предположим, что объекты сплайнов
            // обычно имеют метод reticulate(), который выполняет много
            // вычислений.
            spline.reticulate ? spline.reticulate()
                : spline.reticulated = true;
        }
        // Когда все сплайны (наконец-то!) покрылись сетчатым узором,
        // передать копию обратно главному потоку.
        threads.parentPort.postMessage(splines);
    });
}
```

В первом аргументе конструктора `Worker()` указывается путь к файлу кода JavaScript, который должен быть запущен в потоке. В предыдущем коде мы применяли предварительно определенный идентификатор `__filename` для создания потока воркера, который загружает и запускает тот же файл, что и главный поток. Тем не менее, в общем случае вы будете передавать путь к файлу. Обратите внимание, что если вы указываете относительный путь, то он будет относительным к `process.cwd()`, а не к выполняющемуся в текущий момент модулю. Если вас интересует путь относительно текущего модуля, тогда используйте что-то наподобие `path.resolve(__dirname, 'workers/reticulator.js')`.

Конструктор `Worker()` может также принимать во втором аргументе объект, свойства которого предоставляют необязательную конфигурацию для воркера. Мы раскроем несколько таких параметров позже, но пока отметим, что если вы передаете `{eval: true}` во втором аргументе, то первый аргумент `Worker()` интерпретируется не как имя файла, а как строка кода JavaScript, подлежащего выполнению:

```
new threads.Worker(`
    const threads = require("worker_threads");
    threads.parentPort.postMessage(threads.isMainThread);
`, {eval: true}).on("message", console.log); // Выводится false
```

Среда Node создает копию объекта, переданного `postMessage()`, а не разделяет его непосредственно с потоком воркера, тем самым предотвращая совместное использование памяти потоком воркера и главным потоком. Вы могли бы ожидать, что такое копирование выполняется посредством `JSON.stringify()` и `JSON.parse()` (см. раздел 11.6). Но в действительности среда Node заимствует у веб-браузеров более надежную методику, известную как алгоритм структурированного клонирования.

Алгоритм структурированного клонирования позволяет сериализировать большинство типов JavaScript, включая объекты `Map`, `Set`, `Date` и `RegExp`, а также типизированные массивы, но обычно он не может копировать типы, определенные средой размещения Node, такие как сокеты и потоки. Однако имейте в виду, что объекты `Buffer` частично поддерживаются: если вы передадите `Buffer` методу `postMessage()`, то он будет получен как `Uint8Array` и может быть преобразован обратно в `Buffer` с помощью `Buffer.from()`. Дополнительные сведения об этом алгоритме ищите во врезке "Алгоритм структурированного клонирования" главы 15.

16.11.2. Среда выполнения воркеров

По большей части код JavaScript в потоке воркера Node выполняется в принципе так же, как он выполнялся бы в главном потоке Node. Но существует несколько отличий, о которых следует знать, и некоторые из них касаются свойств необязательного второго аргумента конструктора `Worker()`.

- Как вы видели, свойство `threads.isMainThread` равно `true` в главном потоке, но всегда равно `false` в любом потоке воркера.
- В потоке воркера вы можете применять `threads.parentPort.postMessage()` для отправки сообщения родительскому потоку и `threads.parentPort.on` для регистрации обработчиков событий из родительского потока. В главном потоке `threads.parentPort` всегда будет `null`.
- В потоке воркера `threads.workerData` устанавливается в копию свойства `workerData` второго аргумента конструктора `Worker()`. В главном потоке это свойство всегда равно `null`. Вы можете использовать свойство `workerData` для передачи начального сообщения воркеру, которое будет доступно сразу после его запуска, так что воркеру не придется ожидать события `"message"`, прежде чем он сможет приступить к выполнению своей работы.
- По умолчанию `process.env` в потоке воркера представляет собой копию `process.env` в главном потоке. Но в родительском потоке можно указывать специальный набор переменных среды, устанавливая свойство `env` второго аргумента конструктора `Worker()`. В качестве особого (и потенциально опасного) случая в родительском потоке свойство `env` можно установить в `threads.SHARE_ENV`, что приведет к разделению двумя потоками одного набора переменных среды, когда изменение, внесенное в одном потоке, будет видно в другом.
- По умолчанию поток `process.stdin` в воркере не содержит читабельных данных. Вы можете изменить это, передав `stdin: true` во втором аргументе конструктора `Worker()`, и тогда свойство `stdin` объекта `Worker` будет потоком `Writable`. Любые данные, которые родительский поток записывает в `worker.stdin`, становятся доступными в потоке `process.stdin` внутри воркера.

- По умолчанию потоки `process.stdout` и `process.stderr` в воркере просто соединяются каналами с соответствующими потоками в родительском потоке. Это означает, например, что `console.log()` и `console.error()` производят вывод в потоке воркера точно таким же способом, как в главном потоке. Вы можете переопределить такое поведение, передав конструктору `Worker()` во втором аргументе `stdout:true` или `stderr:true`. В результате любой вывод, который воркер записывает в указанные потоки, становится читабельным для родительского потока через потоки `worker.stdout` и `worker.stderr`. (Здесь присутствует потенциально запутывающая инверсия направлений потоков, и ранее в главе вы видели аналогичную ситуацию с дочерними процессами: потоки выходных данных потока воркера являются просто потоками входных данных родительского потока, а поток входных данных воркера — потоком выходных данных родительского потока.)
- Если в потоке воркера вызывается `process.exit()`, тогда работу завершает только поток, но не весь процесс.
- Потокам воркеров не разрешено изменять разделяемое состояние процесса, к которому они относятся. Функции вроде `process.chdir()` и `process.setuid()` при вызове из воркера генерируют исключение.
- Сигналы ОС (наподобие `SIGINT` и `SIGTERM`) доставляются только главному потоку; они не могут быть получены или обработаны в потоках воркеров.

16.11.3. Каналы связи и объекты `MessagePort`

Когда создается новый поток воркера, вместе с ним создается канал связи, который делает возможной передачу сообщений между потоком воркера и родительским потоком в обоих направлениях. Как вы видели, поток воркера применяет `threads.parentPort` для отправки и получения сообщений в и из родительского потока, а родительский поток использует объект `Worker` для отправки и получения сообщений в и из потока воркера.

API-интерфейс потока воркера также позволяет создавать специальные каналы связи с применением определяемого веб-браузерами API-интерфейса `MessageChannel`, который раскрывался в подразделе 15.13.5. Если вы читали этот подраздел, тогда большинство следующего материала покажется вам знакомым.

Предположим, что воркер должен обрабатывать два разных вида сообщений, посылаемых двумя разными модулями в главном потоке. Такие два модуля могли бы разделять стандартный канал и посылать сообщения посредством `worker.postMessage()`, но было бы лучше, если бы каждый модуль имел собственный закрытый канал для отправки сообщений воркеру. Или возьмем случай, когда главный поток создает два независимых воркера. Специальный канал связи может позволить двум воркерам обмениваться информацией непосредственно друг с другом, не занимаясь отправкой своих сообщений через родительский поток.

Новый канал сообщений создается с помощью конструктора `MessageChannel()`. Объект `MessageChannel` имеет два свойства с именами `port1` и `port2`, которые ссылаются на пару объектов `MessagePort`. Вызов `postMessage()` на одном порту приводит к генерации события `"message"` на другом со структурированным клоном объекта `Message`:

```
const threads = require("worker_threads");
let channel = new threads.MessageChannel();
channel.port2.on("message", console.log); // Вывести любые
                                          // полученные сообщения.
channel.port1.postMessage("hello");       // Приведет к выводу hello.
```

Вы также можете вызвать `close()` на любом порту, чтобы разорвать подключение между двумя портами и сигнализировать о том, что обмена сообщениями больше не будет. Когда `close()` вызывается на любом порту, обоим портам доставляется событие `"close"`.

Обратите внимание, что в приведенном выше примере кода создается пара объектов `MessagePort`, которые затем используются для передачи сообщения внутри главного потока. Для применения специальных каналов связи с воркерами мы должны передать один из двух портов из потока, в котором он создается, в поток, где он будет использоваться. В следующем подразделе объясняется, как это сделать.

16.11.4. Передача объектов `MessagePort` и типизированных массивов

Функция `postMessage()` применяет алгоритм структурированного клонирования, и мы уже отмечали, что он не может копировать объекты вроде сокетов и потоков. Он способен обрабатывать объекты `MessagePort`, но только как особый случай с использованием специальной методики. Метод `postMessage()` (объекта `Worker`, объекта `threads.parentPort` или любого объекта `MessagePort`) принимает необязательный второй аргумент (по имени `transferList`), который представляет собой массив объектов, подлежащих передаче между потоками, а не копированию.

Объект `MessagePort` не может копироваться алгоритмом структурированного клонирования, но его можно передавать. Если первый аргумент `postMessage()` включает один или большее количество объектов `MessagePort` (произвольно глубоко вложенных внутри объекта `Message`), тогда эти объекты `MessagePort` должны также быть элементами массива, передаваемого во втором аргументе. Тем самым среде Node сообщается о том, что ей не нужно делать копию объекта `MessagePort`, а взамен можно просто предоставить другому потоку существующий объект. Тем не менее, в отношении передачи значений между потоками важно понимать, что после того, как значение передано, его больше нельзя применять в потоке, который вызвал `postMessage()`.

Вот как можно создать новый объект `MessageChannel` и передать воркеру один из его объектов `MessagePort`:

```
// Создать специальный канал связи.
const threads = require("worker_threads");
let channel = new threads.MessageChannel();

// Использовать стандартный канал воркера для передачи воркеру
// одного конца нового канала. Предполагается, что когда воркер
// получает это сообщение, он немедленно начинает прослушивать
// сообщения на новом канале.
worker.postMessage({ command: "changeChannel", data: channel.port1 },
                   [ channel.port1 ]);

// Послать сообщение воркеру, используя конец специального канала.
channel.port2.postMessage("Can you hear me now?");

// И также прослушивать ответы от воркера.
channel.port2.on("message", handleMessagesFromWorker);
```

Объекты `MessagePort` — не единственные объекты, которые можно передавать. Если вы вызываете `postMessage()` с типизированным массивом в качестве сообщения (или с сообщением, которое содержит один и более типизированных массивов, произвольно глубоко вложенных внутри сообщения), то этот типизированный массив (или типизированные массивы) будут просто копироваться алгоритмом структурированного клонирования. Но типизированные массивы могут быть большими, скажем, если вы используете поток воркера для обработки изображений, включающих миллионы пикселей. Таким образом, ради эффективности `postMessage()` также дает нам возможность передавать типизированные массивы вместо их копирования. (Потоки по умолчанию разделяют память. Потоки воркеров в JavaScript, как правило, избегают разделения памяти, но когда мы разрешаем такую контролируемую передачу, то она может выполняться очень эффективно.) Безопасным подход делает то, что после передачи типизированного массива в другой поток он становится неиспользуемым в потоке, который его передал. В сценарии с обработкой изображений главный поток может передать пиксели изображения потоку воркера, после чего поток воркера может передать обработанные пиксели обратно главному потоку. Память копировать не нужно, но она никогда не будет доступной сразу двум потокам.

Чтобы передать типизированный массив вместо его копирования, включите объект `ArrayBuffer`, поддерживающий массив, во второй аргумент `postMessage()`:

```
let pixels = new Uint32Array(1024*1024);  // 4 мегабайта памяти.

// Предположим, что мы прочитали данные в типизированный массив pixels
// и затем передаем pixels воркеру без копирования. Обратите внимание,
// что мы помещаем в список передачи не сам массив, а объект Buffer
// массива.
worker.postMessage(pixels, [ pixels.buffer ]);
```

Как и переданные объекты `MessagePort`, переданные типизированные массивы становятся неиспользуемыми после передачи. При попытке работы с объектом `MessagePort` или типизированным массивом, который был передан, никакие исключения не генерируются; эти объекты просто перестают делать что-либо, когда вы с ними взаимодействуете.

16.11.5. Разделение типизированных массивов между потоками

Кроме передачи типизированных массивов из потока в поток типизированные массивы можно разделять между потоками. Просто создайте буфер `SharedArrayBuffer` с желаемым размером и примените его для создания типизированного массива. Когда типизированный массив, поддерживаемый `SharedArrayBuffer`, передается посредством `postMessage()`, лежащая в основе память будет разделяться между потоками. В таком случае вы не должны включать разделяемый буфер во второй аргумент `postMessage()`.

Однако на самом деле вам не следует поступать так, поскольку язык JavaScript никогда не задумывался как безопасный в отношении потоков, а при многопоточном программировании очень трудно добиться правильных результатов. (Именно потому объект `SharedArrayBuffer` не рассматривался в разделе 11.2: он представляет собой узкоспециализированное средство, с которым нелегко обращаться.) Даже простая операция `++` не безопасна к потокам, т.к. ей необходимо прочитать значение, инкрементировать его и записать обратно. Если два потока одновременно инкрементируют значение, то зачастую оно будет инкрементировано только один раз, как демонстрируется в следующем коде:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    // В главном потоке мы создаем разделяемый типизированный массив
    // с одним элементом. Оба потока будут иметь возможность читать
    // и записывать sharedArray[0] в одно и то же время.
    let sharedBuffer = new SharedArrayBuffer(4);
    let sharedArray = new Int32Array(sharedBuffer);

    // Теперь создадим поток воркера, передавая ему разделяемый массив
    // как начальное значение workerData, чтобы не беспокоиться
    // об отправке и получении сообщения.
    let worker =
        new threads.Worker(__filename, { workerData: sharedArray });

    // Ожидаем, пока воркер начнет выполняться, и затем инкрементируем
    // разделяемое целое число 10 миллионов раз.
    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;

        // Закончив инкрементирование, мы начинаем прослушивать события
        // "message", чтобы знать, когда воркер завершит работу.
        worker.on("message", () => {
            // Хотя разделяемое целое число было инкрементировано
            // 20 миллионов раз, его значение в общем случае будет
            // намного меньше. На моем компьютере финальное значение
            // обычно оказывается ниже 12 миллионов.
            console.log(sharedArray[0]);
        });
    });
} else {
```

```
    // В потоке воркера мы получаем разделяемый массив из workerData
    // и затем инкрементируем его 10 миллионов раз.
    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) sharedArray[0]++;
    // Уведомляем главный поток об окончании инкрементирования.
    threads.parentPort.postMessage("done");
}
```

Один сценарий, в котором использование `SharedArrayBuffer` может иметь смысл, касается ситуации, когда два потока оперируют с полностью отдельными сегментами разделяемой памяти. Вы можете обеспечить это, создав два типизированных массива, которые будут служить представлениями неперекрывающихся областей разделяемого буфера, и затем заставить два потока работать с созданными двумя отдельными типизированными массивами. Для выполнения параллельной сортировки слиянием один поток мог бы сортировать нижнюю половину массива, а другой поток — верхнюю. Такой подход также окажется подходящим при реализации определенных видов алгоритмов обработки изображений: множество потоков будут работать с непересекающимися областями изображения.

Если вам действительно нужно разрешить множеству потоков иметь доступ к одной и той же области разделяемого массива, тогда вы можете сделать шаг в сторону безопасности к потокам за счет применения функций, определенных в объекте `Atomics`. Объект `Atomics` был добавлен в JavaScript, когда объект `SharedArrayBuffer` должен был определить атомарные операции над элементами разделяемого массива. Например, функция `Atomics.add()` читает указанный элемент разделяемого массива, добавляет к нему заданное значение и записывает сумму обратно в массив. Она делает это атомарно, как если бы была одиночной операцией, и гарантирует, что во время выполнения операции никакой другой поток не сможет читать или записывать значение. Функция `Atomics.add()` дает нам возможность переписать показанный ранее код параллельного инкрементирования и получить корректный результат в 20 миллионов инкрементов элемента разделяемого массива:

```
const threads = require("worker_threads");

if (threads.isMainThread) {
    let sharedBuffer = new SharedArrayBuffer(4);
    let sharedArray = new Int32Array(sharedBuffer);
    let worker =
        new threads.Worker(__filename, { workerData: sharedArray });

    worker.on("online", () => {
        for(let i = 0; i < 10_000_000; i++) {
            Atomics.add(sharedArray, 0, 1);    // Безопасный к потокам
                                               // атомарный инкремент.
        }

        worker.on("message", (message) => {
            // Когда оба потока завершают работу,
            // использовать безопасную к потокам
            // функцию для чтения элемента разделяемого массива и
            // подтверждения, что он имеет ожидаемое значение 20000000.
```

```
            console.log(Atomics.load(sharedArray, 0));
        });
    });
} else {
    let sharedArray = threads.workerData;
    for(let i = 0; i < 10_000_000; i++) {
        Atomics.add(sharedArray, 0, 1);        // Безопасный к потокам
                                               // атомарный инкремент.
    }
    threads.parentPort.postMessage("done");
}
```

Приведенная новая версия кода выводит правильное число 20 000 000. Тем не менее, она приблизительно в девять раз медленнее предыдущей некорректной версии кода. Было бы намного проще и быстрее делать все 20 миллионов инкрементов в одном потоке. Также имейте в виду, что атомарные операции могут обеспечить безопасность в отношении потоков алгоритмам обработки изображений, в которых каждый элемент массива является значением, полностью независимым от всех остальных значений. Однако в большинстве реальных программ многие элементы массива часто связаны друг с другом и требуется некоторая синхронизация потоков на более высоком уровне. В таком случае могут помочь низкоуровневые функции `Atomics.wait()` и `Atomics.notify()`, но их обсуждение выходит за рамки настоящей книги.

16.12. Резюме

Хотя язык JavaScript создавался для выполнения кода в веб-браузерах, среда Node превратила JavaScript в универсальный язык программирования. Он особенно популярен в сфере реализации веб-серверов, но его глубокая привязка к ОС означает, что JavaScript также является хорошей альтернативой сценариям оболочки. Ниже перечислены важные темы, которые были раскрыты в этой длинной главе.

- Асинхронные по умолчанию API-интерфейсы среды Node, а также ее однопоточный стиль параллелизма, основанный на обратных вызовах и событиях.
- Фундаментальные типы данных, буферы и потоки Node.
- Модули `fs` и `path` среды Node, предназначенные для работы с файловой системой.
- Модули `http` и `https` среды Node, предназначенные для написания клиентов и серверов HTTP.
- Модуль `net` среды Node, предназначенный для написания клиентов и серверов, которые не работают с HTTP.
- Модуль `child_process` среды Node, предназначенный для создания и взаимодействия с дочерними процессами.
- Модуль `worker_threads` среды Node, предназначенный для подлинного многопоточного программирования с использованием передачи сообщений, а не разделяемой памяти.

ГЛАВА 17

Инструменты и расширения JavaScript

Примите поздравления — вы добрались до последней главы настоящей книги! Если вы прочитали все, что было написано ранее, то теперь хорошо разбираетесь в языке JavaScript и знаете, как его использовать в Node и в веб-браузерах. Эта глава является окончанием: в ней рассмотрен ряд важных программных инструментов, которые многие программисты на JavaScript считают полезными, и также описаны два широко применяемых расширения базового языка JavaScript. Независимо от того, решите вы использовать такие расширения в своих проектах или нет, они почти наверняка встретятся вам в других проектах, а потому важно хотя бы знать, что они собой представляют.

Ниже перечислены инструменты и языковые расширения, которые будут раскрываться далее в главе.

- ESLint для поиска потенциальных дефектов и стилевых проблем в вашем коде.
- Prettier для форматирования кода JavaScript в стандартизованной манере.
- Jest как универсальное решение для написания модульных тестов JavaScript.
- npm для управления и установки программных библиотек, от которых зависит программа.
- Инструменты пакетирования кода, подобные webpack, Rollup и Parcel, которые преобразуют ваши модули кода JavaScript в единственный пакет для применения в веб-сети.
- Babel для трансляции кода JavaScript, использующего совершенно новые языковые средства (или языковые расширения), в код JavaScript, который может выполняться в текущих веб-браузерах.

- Языковое расширение JSX (применяемое фреймворком React), которое позволяет описывать пользовательские интерфейсы с использованием выражений JavaScript, выглядящих как HTML-разметка.
- Языковое расширение Flow (или похожее расширение TypeScript), которое позволяет аннотировать код JavaScript с помощью типов и проверять код на предмет безопасности в отношении типов.

В главе не приводится детальное описание упомянутых инструментов и расширений. Цель заключается в том, чтобы просто объяснить их достаточно подробно, дабы вы смогли понять, почему они полезны и когда имеет смысл их применять. Все, что обсуждается в данной главе, широко используется в мире программирования на JavaScript, и если вы решите задействовать какой-то инструмент или расширение, то найдете в Интернете множество документации и руководств.

17.1. Линтинг с помощью ESLint

В программировании термин *линт* (lint) относится к коду, который будучи формально корректным, является неприглядным, содержит возможный дефект или субоптимален в некотором смысле. *Линтер* (linter; статический анализатор) — это инструмент для обнаружения линта в коде, а *линтинг* (linting) — процесс запуска линтера для вашего кода (и последующее исправление кода в целях устранения линта, чтобы линтер больше не сообщал о нем).

Самым часто применяемым линтером для JavaScript на сегодняшний день считается ESLint (`https://eslint.org`). Если вы запустите его и затем уделите время устранению проблем, на которые он укажет, то сделаете свой код яснее и снизите вероятность наличия в нем дефектов. Взгляните на следующий код:

```
var x = 'unused';

export function factorial(x) {
   if (x == 1) {
     return 1;
   } else {
      return x * factorial(x-1)
   }
}
```

Запустив ESLint для показанного кода, вы можете получить вывод такого вида:

```
$ eslint code/ch17/linty.js

code/ch17/linty.js
  1:1   error    Unexpected var, use let or const instead      no-var
        ошибка   Непредвиденное ключевое слово var,
                 используйте взамен let или const
  1:5   error    'x' is assigned a value but never used    no-unused-vars
        ошибка   x присваивается значение, но никогда не используется
  1:9   warning  Strings must use doublequote                  quotes
        предупреждение  В строках должны использоваться двойные кавычки
```

```
4:11  error    Expected '===' and instead saw '=='      eqeqeq
      ошибка   Ожидалась операция ===, а взамен обнаружена ==
5:1   error    Expected indentation of 8 spaces but found 6  indent
      ошибка   Ожидался отступ в 8 пробелов, а взамен обнаружено 6
7:28  error    Missing semicolon                        semi
      ошибка   Не хватает точки с запятой

✖ 6 problems (5 errors, 1 warning)
  6 проблем (5 ошибок, 1 предупреждение)

  3 errors and 1 warning potentially fixable with the `--fix` option.
  3 ошибки 1 предупреждение потенциально исправимы с помощью параметра `--fix`
```

Иногда линтеры могут показаться придирчивыми. Действительно ли имеет значение, какие кавычки мы применяем для строк — двойные или одинарные? С другой стороны, правильное использование отступов важно для читабельности, тогда как применение `===` и `let` вместо `==` и `var` защищает от тонких ошибок. А неиспользуемые переменные являются мертвым грузом в вашем коде — нет никаких причин держать их под рукой.

ESLint определяет многие правила линтинга и имеет экосистему подключаемых модулей, которые добавляют множество дополнительных правил. Но инструмент ESLint полностью конфигурируемый и вы можете создать конфигурационный файл, который настроит ESLint для применения только тех правил, которые вас интересуют.

17.2. Форматирование кода JavaScript с помощью Prettier

Одна из причин использования линтеров в ряде проектов связана с необходимостью обеспечения согласованного стиля написания кода, чтобы команда программистов, работающая над разделяемой кодовой базой, применяла совместимые соглашения по оформлению кода. Сюда входят не только правила отступа кода, но и такие вещи, как предпочтительный вид кавычек и наличие пробела между ключевым словом `for` и следующей за ним открывающей круглой скобкой.

Современная альтернатива навязыванию правил форматирования кода через линтер предусматривает использование инструмента вроде Prettier (`https://prettier.io`) для автоматического синтаксического анализа и переформатирования всего кода.

Предположим, что вы написали показанную ниже функцию, которая работает, но ее код сформатирован нестандартно:

```
function factorial(x)
{
       if(x===1){return 1}
         else{return x*factorial(x-1)}
}
```

Запуск Prettier для этого кода исправляет отступы, добавляет недостающие точки с запятой, добавляет пробелы вокруг бинарных операций и вставляет разрывы строк после { и перед }, давая в результате гораздо более стандартно выглядящий код:

```
$ prettier factorial.js
function factorial(x) {
  if (x === 1) {
    return 1;
  } else {
    return x * factorial(x - 1);
  }
}
```

Если вы вызовите инструмент Prettier с параметром `--write`, тогда он будет просто переформатировать указанный файл на месте, а не выводить переформатированную версию. В случае применения `git` для управления исходным кодом вы можете вызывать Prettier с параметром `--write` в привязке фиксации, чтобы код автоматически форматировался перед помещением в хранилище.

Инструмент Prettier особенно эффективен, если вы сконфигурируете свой редактор для автоматического запуска Prettier при каждом сохранении вами файла. Я чувствую себя свободным, когда пишу небрежный код и вижу, как он для меня автоматически корректируется.

Prettier допускает конфигурирование, но имеет совсем немного параметров. Вы можете указывать максимальную длину строки, размер отступа, должны ли использоваться точки с запятой, в какие кавычки должны помещаться строки (одинарные или двойные) и ряд других аспектов. В целом стандартные параметры Prettier вполне разумны. Идея в том, что вы лишь задействуете Prettier в своем проекте, после чего вам больше никогда не придется снова думать о форматировании кода.

Лично мне по-настоящему нравится применять инструмент Prettier в проектах на JavaScript. Однако я не использовал его для кода в этой книге, т.к. в большей части кода я опирался на аккуратное форматирование вручную, чтобы выравнивать свои комментарии по вертикали, а Prettier привел бы их в беспорядок.

17.3. Модульное тестирование с помощью Jest

Написание тестов — важная часть любого нетривиального программного проекта. Динамические языки наподобие JavaScript поддерживают фреймворки для тестирования, которые значительно сокращают усилия, требующиеся для написания тестов, и делают реализацию тестов почти забавной! Существует множество инструментов и библиотек для тестирования кода JavaScript, и многие из них написаны по модульному принципу, так что одну библиотеку можно выбрать как средство прогона тестов, другую для утверждений и третью для имитации. Тем не менее, в этом разделе рассматривается Jest (`https://jestjs.io`) — популярный фреймворк, который включает все необходимое в единственном пакете.

Предположим, что вы написали следующую функцию:

```
const getJSON = require("./getJSON.js");

/**
 * Функция getTemperature() принимает на входе название города
 * и возвращает объект Promise, который будет разрешен в текущую
 * температуру в этом городе, выраженную в градусах по Фаренгейту.
 * Она полагается на (фиктивную) веб-службу,
 * которая возвращает мировую температуру в градусах по Цельсию.
 */
module.exports = async function getTemperature(city) {
    // Получить температуру в градусах по Цельсию от веб-службы.
    let c = await getJSON(
        `https://globaltemps.example.com/api/city/${city.toLowerCase()}`
    );
    //Преобразовать ее в градусы по Фаренгейту и возвратить это значение
    return (c * 5 / 9) + 32;  // Что сделать: дважды проверьте эту формулу
};
```

Хороший набор тестов для функции `getTemperature()` может проверять, что она извлекает данные из правильного URL и корректно выполняет преобразование между температурными шкалами. Мы можем достичь цели с помощью приведенного ниже теста, основанного на Jest, где определяется имитированная реализация `getJSON()`, так что тест фактически не осуществляет сетевой запрос. И поскольку `getTemperature()` является асинхронной функцией, тесты также асинхронны — тестировать асинхронные функции может быть сложно, но Jest делает процесс относительно легким:

```
// Импортировать функцию, которую мы собираемся тестировать.
const getTemperature = require("./getTemperature.js");

// Имитировать модуль getJSON(), от которого зависит getTemperature().
jest.mock("./getJSON");
const getJSON = require("./getJSON.js");

// Сообщить имитированной функции getJSON() о необходимости возврата
// уже разрешенного объекта Promise со значением разрешения 0.
getJSON.mockResolvedValue(0);

// Здесь начинается наш набор тестов для getTemperature().
describe("getTemperature()", () => {
    // Это первый тест. Мы удостоверяемся в том, что getTemperature()
    // вызывает getJSON() с ожидаемым URL.
    test("Invokes the correct API", async () => {
        // Вызывает корректный API-интерфейс.
    let expectedURL =
                "https://globaltemps.example.com/api/city/vancouver";
        let t = await(getTemperature("Vancouver"));
        // Имитированные реализации Jest запоминают, как они вызывались,
        // и мы можем это проверить.
        expect(getJSON).toHaveBeenCalledWith(expectedURL);
    });
```

```
  // Второй тест проверяет, что getTemperature() корректно
  // преобразует градусы по Цельсию в градусы по Фаренгейту.
  test("Converts C to F correctly", async () => {// Корректно
                                                  // преобразует C в F
    getJSON.mockResolvedValue(0);    // Если getJSON возвращает 0C,
    expect(await getTemperature("x")).toBe(32); //то мы ожидаем 32F

    // 100C должно преобразовываться в 212F.
    getJSON.mockResolvedValue(100); // Если getJSON возвращает 100C,
    expect(await getTemperature("x")).toBe(212); //то мы ожидаем 212F
  });
});
```

Имея написанные тесты, мы можем применить команду `jest` для их прогона и обнаружить, что один из тестов не проходит:

```
$ jest getTemperature
FAIL ch17/getTemperature.test.js
 getTemperature()
   ✓ Invokes the correct API (4ms)
   ✓ Вызывает корректный API-интерфейс (4 мс)
   ✕ Converts C to F correctly (3ms)
   ✕ Корректно преобразует C в F (3 мс)

 ● getTemperature() › Converts C to F correctly

   expect(received).toBe(expected) // Object.is equality

   Expected: 212
   Received: 87.55555555555556

     29 |       // 100C должно преобразовываться в 212F.
     30 |       getJSON.mockResolvedValue(100);  // Если getJSON
                                                 // возвращает 100C,
   > 31 |       expect(await getTemperature("x")).toBe(212);  // то мы
                                                 // ожидаем 212F.
        |                                          ^
     32 |     });
     33 | });
     34 |

     at Object.<anonymous> (ch17/getTemperature.test.js:31:43)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.403s
Ran all test suites matching /getTemperature/i.
Тестовые комплекты: 1 не прошел, всего 1
Тесты:              1 не прошел, 1 прошел, всего 2
Снимки:             всего 0
Время:              1.403 с
Выполнены все тестовые комплекты, соответствующие /getTemperature/i.
```

В нашей реализации `getTemperature()` используется неправильная формула для преобразования градусов по Цельсию в градусы по Фаренгейту. Она умножает на 5 и делит на 9, вместо того чтобы умножать на 9 и делить на 5. Если

мы исправим код и запустим Jest еще раз, то увидим, что тесты проходят. И в качестве бонуса, если при запуске `jest` мы добавим параметр `--coverage`, то инструмент рассчитает и отобразит покрытие кода нашими тестами:

```
$ jest --coverage getTemperature
PASS ch17/getTemperature.test.js
 getTemperature()
   ✓ Invokes the correct API (3ms)
   ✓ Converts C to F correctly (1ms)

------------------|---------|---------|---------|---------|-----------------
File              | % Stmts | %Branch | % Funcs |  % Lines| Uncovered Line #s
------------------|---------|---------|---------|---------|-----------------
All files         |   71.43 |     100 |   33.33 |   83.33 |
 getJSON.js       |   33.33 |     100 |       0 |      50 |               2
 getTemperature.js|     100 |     100 |     100 |     100 |
------------------|---------|---------|---------|---------|-----------------
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.508s
Ran all test suites matching /getTemperature/i.
```

Прогон наших тестов дал нам 100%-ное покрытие кода для протестированного модуля, к чему мы и стремились. Они лишь частично покрыли `getJSON()`, но мы имитировали данный модуль и не пытались его тестировать, так что все ожидаемо.

17.4. Управление пакетами с помощью npm

В современной разработке программного обеспечения любая нетривиальная программа, которую вы пишете, обычно зависит от сторонних программных библиотек. Например, если вы реализуете веб-сервер в Node, то могли бы применять фреймворк Express. А если вы создаете пользовательский интерфейс для отображения в веб-браузере, то могли бы использовать фреймворк вроде React, LitElement или Angular. Диспетчер пакетов облегчает поиск и установку сторонних пакетов такого рода. Не менее важно то, что диспетчер пакетов отслеживает, от каких пакетов зависит ваш код, и сохраняет эту информацию в файле. В результате, когда кто-то другой пожелает испытать вашу программу, он может загрузить ее код и список зависимостей, после чего с помощью собственного диспетчера пакетов установить все сторонние пакеты, в которых нуждается ваш код.

В состав Node входит диспетчер пакетов `npm`, который был представлен в подразделе 16.1.5. Он в равной степени полезен как при программировании на JavaScript стороны клиента, так и при программировании на стороне сервера посредством Node.

Если вы испытываете чей-то проект JavaScript, то первое, что вы часто делаете после загрузки кода — вводите команду `npm install`. Она читает зависимости, перечисленные в файле `package.json`, загружает сторонние пакеты, необходимые проекту, и сохраняет их в каталоге `node_modules/`.

Для установки определенного пакета в каталоге node_modules/ вашего проекта введите команду npm install <имя-пакета>:

```
$ npm install express
```

Помимо установки указанного пакета npm также создает запись о зависимости в файле package.json для проекта. Регистрация зависимостей подобным образом позволяет остальным устанавливать эти зависимости, просто вводя npm install.

Другой вид зависимости — инструменты разработчика, которые нужны разработчикам, желающим трудиться над вашим проектом, но для выполнения кода они не требуются. Скажем, если в проекте применяется Prettier для обеспечения согласованного форматирования всего кода, тогда Prettier будет "зависимостью разработки", которую вы можете установить и зарегистрировать с помощью параметра --save-dev:

```
$ npm install --save-dev prettier
```

Иногда у вас может возникать потребность установить инструменты разработчика глобально, чтобы они были доступными где угодно в коде, который формально не входит в состав проекта с файлом package.json и каталогом node_modules/. В таком случае вы можете использовать параметр -g (global — глобальная):

```
$ npm install -g eslint jest
/usr/local/bin/eslint -> /usr/local/lib/node_modules/eslint/bin/eslint.js
/usr/local/bin/jest -> /usr/local/lib/node_modules/jest/bin/jest.js
+ jest@24.9.0
+ eslint@6.7.2
added 653 packages from 414 contributors in 25.596s

$ which eslint
/usr/local/bin/eslint
$ which jest
/usr/local/bin/jest
```

В дополнение к команде install диспетчер npm поддерживает команды uninstall и update, которые выполняют удаление и обновление. В npm также есть интересная команда audit, которую вы можете применять для поиска в своих зависимостях уязвимостей, относящихся к безопасности, и их устранения:

```
$ npm audit --fix
                              === npm audit security report ===
found 0 vulnerabilities
 in 876354 scanned packages
```

Когда вы устанавливаете инструмент наподобие ESLint локально для проекта, сценарий eslint попадает в ./node_modules/.bin/eslint, что делает команду неудобной для запуска. К счастью, диспетчер npm связан с командой npx, которую вы можете использовать для запуска локально установленных инструментов с помощью команды вида npx eslint или npx jest. (А если вы применяете команду npx для вызова инструмента, который еще не был установлен, тогда она установит его.)

Компания, стоящая за npm, также поддерживает хранилище пакетов https://npmjs.com, где находятся сотни тысяч пакетов с открытым кодом. Но вы не обязаны использовать диспетчер пакетов npm для доступа к указанному хранилищу пакетов. Альтернативы включают yarn (https://yarnpkg.com) и pnpm (https://pnpm.js.org).

17.5. Пакетирование кода

Если вы пишете крупную программу на JavaScript для запуска в веб-браузерах, то вероятно захотите применять какой-нибудь инструмент пакетирования кода, особенно когда в программе используются внешние пакеты, которые доставляются в виде модулей. Разработчики веб-приложений применяли модули ES6 (см. раздел 10.3) на протяжении многих лет, задолго до того, как в веб-сети начали поддерживаться ключевые слова import и export. В такой ситуации программисты используют инструмент пакетирования кода, который начинает с главной точки входа (или точек входа) программы и следует по дереву директив import для нахождения всех модулей, от которых зависит программа. Затем он объединяет все индивидуальные файлы модулей в единый пакет кода JavaScript и переписывает директивы import и export, чтобы заставить код работать в такой новой форме. Результатом является единственный файл кода, который может быть загружен в веб-браузер, не поддерживающий модули.

В наши дни модули ES6 поддерживаются веб-браузерами почти повсеместно, но разработчики веб-приложений по-прежнему склонны применять инструменты пакетирования кода, во всяком случае, при выпуске производственного кода. Разработчики считают, что пользователям будет удобнее, если при первом посещении веб-сайта загружается один пакет кода среднего размера, чем когда загружается много мелких модулей.

Веб-производительность — как известно, сложная тема с огромным числом переменных, которые необходимо учитывать, включая непрерывные усовершенствования со стороны создателей веб-браузеров, поэтому единственный способ быть уверенным в самом быстром методе загрузки вашего кода предусматривает основательное тестирование и тщательное измерение. Имейте в виду, что есть одна переменная, которая находится полностью под вашим контролем: размер кода. Меньший объем кода JavaScript всегда будет загружаться быстрее, чем больший объем кода JavaScript!

Доступно несколько хороших инструментов пакетирования кода JavaScript. В число самых часто используемых инструментов пакетирования входят webpack (https://webpack.js.org), Rollup (https://rollupjs.org/guide/en) и Parcel (https://parceljs.org). Базовые возможности инструментов пакетирования более или менее одинаковы, и они различаются в зависимости от того, до какой степени допускают конфигурирование и насколько просты в эксплуатации. Инструмент webpack существует уже давно, обладает большой экосисте-

мой подключаемых модулей, допускает расширенное конфигурирование и способен поддерживать более старые немодульные библиотеки. Но он также может быть сложен и труден в настройке. На другом конце спектра располагается инструмент Parcel, который задуман как альтернатива с нулевой конфигурацией, просто делающая то, что нужно.

Помимо выполнения базового пакетирования инструменты пакетирования также предлагают ряд дополнительных средств.

- Некоторые программы имеют более одной точки входа. Например, вебприложение с множеством страниц может быть написано так, что в нем предусмотрена отдельная точка входа для каждой страницы. Инструменты пакетирования обычно позволяют создавать один пакет на точку входа или единственный пакет, который поддерживает множество точек входа.
- Программы могут применять `import()` в функциональной форме (см. подраздел 10.3.6) вместо статической, чтобы динамически загружать модули, когда они действительно необходимы, а не загружать их статически при запуске программы. Такой прием часто позволяет сократить время запуска программ. Инструменты пакетирования, которые поддерживают `import()`, могут быть способны производить множество выходных пакетов: одного для загрузки во время запуска и одного и более для динамической загрузки по мере надобности. Это может хорошо работать, если в программе есть лишь несколько вызовов `import()` и они загружают модули с относительно непересекающимися наборами зависимостей. Если динамически загружаемые модули разделяют зависимости, тогда становится сложно выяснить, сколько пакетов нужно произвести, и для решения проблемы вам видимо придется вручную конфигурировать свой инструмент пакетирования.
- Инструменты пакетирования, как правило, могут выводить файл исходной карты (source map), в котором определено сопоставление между строками кода в пакете и соответствующими строками в первоначальных файлах исходного кода. Он позволяет инструментам разработчика в браузере автоматически отображать сообщения об ошибках JavaScript в их первоначальных непакетированных местоположениях.
- Временами, когда вы импортируете модуль в свою программу, то используете только немногие его средства. Хороший инструмент пакетирования способен анализировать код с целью определения, какие его части не задействованы и могут быть опущены из пакетов. Такое средство получило причудливое название "метод для тряски дерева".
- Инструменты пакетирования обычно имеют основанную на подключаемых модулях архитектуру и поддерживают подключаемые модули, позволяющие импортировать и пакетировать "модули", которые в действительности не являются файлами кода JavaScript. Предположим, что ваша программа содержит крупную структуру данных, совместимую с JSON. Инструменты пакетирования кода можно сконфигурировать, чтобы сде-

лать возможным перемещение такой структуры данных в отдельный файл JSON и затем его импортирование в программу с помощью объявления вроде `import widgets from "./big-widget-list.json"`. Аналогично разработчики веб-приложений, встраивающие CSS-стили в свои программы JavaScript, могут применять подключаемые модули инструмента пакетирования, которые позволят им импортировать файлы CSS посредством директивы `import`. Однако имейте в виду, что если вы импортируете что-то отличающееся от файла кода JavaScript, то используете нестандартное расширение JavaScript и делаете свой код зависимым от инструмента пакетирования.

- В языке, подобном JavaScript, который не требует компиляции, запуск инструмента пакетирования выглядит подобно шагу компиляции, и запускать его после каждого редактирования кода перед выполнением кода в браузере утомительно. Инструменты пакетирования, как правило, поддерживают наблюдатели файловой системы, которые обнаруживают редактирование любых файлов в каталоге проекта и автоматически заново генерируют необходимые пакеты. С применением такого средства вы обычно можете сохранять свой код и затем немедленно перегружать окно веб-браузера, чтобы испытать его.
- Некоторые инструменты пакетирования также поддерживают для разработчиков режим "горячей замены модулей", при котором каждый раз, когда пакет генерируется повторно, он автоматически загружается в браузер. Если упомянутый режим работает, тогда обеспечивает сверхъестественное удобство для разработчиков, но "за кулисами" есть ряд трюков, которые придется предпринять, чтобы заставить его работать, и он подходит не для всех проектов.

17.6. Транспиляция с помощью Babel

Babel — это инструмент, который компилирует код JavaScript, написанный с использованием современных языковых средств, в код JavaScript, где такие современные языковые средства не применяются. Поскольку инструмент Babel компилирует код JavaScript в код JavaScript, его иногда называют "транспилятором" (transpiler). Инструмент Babel был создан для того, чтобы разработчики веб-приложений могли использовать новые языковые средства ES6 и последующих версий, по-прежнему ориентируясь на веб-браузеры, которые поддерживают только ES5.

Такие языковые средства, как операция возведения в степень `**` и стрелочные функции, могут относительно легко трансформироваться в выражения `Math.pow()` и `function`. Другие языковые средства вроде ключевого слова `class` требуют гораздо более сложных трансформаций, и в общем случае код, выдаваемый инструментом Babel, не предназначен для чтения человеком. Тем не менее, подобно инструментам пакетирования Babel может производить исходные карты, которые сопоставляют позиции в трансформированном коде с

позициями в первоначальном исходном коде, что оказывает значительную помощь при работе с трансформированным кодом.

Поставщикам браузеров неплохо удается угнаться за эволюцией языка JavaScript, и в наши дни существует намного меньшая потребность в избавлении от стрелочных функций и объявлений классов. Инструмент Babel все еще может помочь, когда вы хотите применять самые последние языковые средства, такие как разделители в виде подчеркиваний в числовых литералах.

Как и большинство остальных инструментов, описанных в главе, вы можете установить Babel посредством npm и запускать его с помощью npx. Инструмент Babel читает конфигурационный файл `.babelrc`, который сообщает ему, каким образом желательно трансформировать ваш код. В Babel определены "предварительные установки", которые вы можете выбирать в зависимости от того, какие языковые расширения хотите использовать и насколько активно трансформировать стандартные языковые средства. Одна из интересных предварительных установок Babel предназначена для сжатия кода путем минификации (удаления комментариев и пробельных символов, переименования переменных и т.д.).

Если вы применяете Babel и какой-то инструмент пакетирования кода, то можете настроить инструмент пакетирования на автоматический запуск Babel для ваших файлов JavaScript, когда он строит пакет. В результате упрощается процесс выпуска исполняемого кода. Скажем, webpack поддерживает модуль `babel-loader`, который вы можете установить и сконфигурировать для запуска Babel с каждым модулем JavaScript по мере его пакетирования.

Несмотря на то что потребность в трансформации базового языка JavaScript в настоящее время снизилась, инструмент Babel по-прежнему широко используется для поддержки нестандартных языковых расширений, два из которых будут описаны далее в главе.

17.7. JSX: выражения разметки в JavaScript

JSX представляет собой расширение базового JavaScript, которое применяет синтаксис в стиле HTML для определения дерева элементов. Расширение JSX наиболее тесно связано с фреймворком React для построения пользовательских интерфейсов веб-приложений. В React деревья элементов, определяемые посредством JSX, в итоге визуализируются веб-браузером как HTML-разметка. Даже если вы сами не планируете использовать фреймворк React, его высокая популярность означает, что вам наверняка встретится код, в котором применяется JSX. В этом разделе объясняется, что необходимо знать для его понимания. (Раздел посвящен не React, а языковому расширению JSX, и фреймворк React рассматривается лишь в том объеме, которого достаточно, чтобы обеспечить контекст для синтаксиса JSX.)

Вы можете считать элемент JSX новым типом синтаксиса выражений JavaScript. Строковые литералы JavaScript определяются с помощью кавычек, а литералы регулярных выражений — посредством символов косой черты. В сходной манере литералы выражений JSX определяются с использованием угловых скобок.

Вот очень простой пример:

```
let line = <hr/>;
```

Если вы применяете JSX, то должны использовать Babel (или аналогичный инструмент) для компиляции выражений JSX в обыкновенный код JavaScript. Трансформация достаточно проста, поэтому некоторые разработчики предпочитают применять React, не используя JSX. Инструмент Babel трансформирует выражение JSX из показанного выше оператора присваивания в простой вызов функции:

```
let line = React.createElement("hr", null);
```

Синтаксис JSX похож на синтаксис HTML, и подобно элементам HTML элементы React могут иметь атрибуты, например:

```
let image = <img src="logo.png" alt="The JSX logo" hidden/>;
```

Когда элемент имеет один или большее количество атрибутов, они становятся свойствами объекта, передаваемого во втором аргументе функции `createElement()`:

```
let image = React.createElement("img", {
            src: "logo.png",
            alt: "The JSX logo",
            hidden: true
});
```

Подобно элементам HTML элементы JSX могут содержать строки и другие элементы в качестве дочерних. Так же, как арифметические операции JavaScript могут применяться для записи арифметических выражений произвольной сложности, элементы JSX можно вкладывать друг в друга произвольно глубоко, создавая деревья элементов:

```
let sidebar = (
  <div className="sidebar">
    <h1>Title</h1>
    <hr/>
    <p>This is the sidebar content</p>
  </div>
);
```

Выражения вызова функций JavaScript тоже могут быть произвольно глубоко вложенными, и приведенные выше вложенные выражения JSX транслируются в набор вложенных вызовов `createElement()`. Если элемент JSX имеет дочерние элементы (обычно строки и другие элементы JSX), тогда они передаются в третьем и последующих аргументах:

```
let sidebar = React.createElement(
  "div", { className: "sidebar"}, // Этот внешний вызов создает <div>.
  React.createElement("h1", null, // Это первый дочерний элемент <div/>
            "Title"),   // и его собственный первый дочерний элемент.
  React.createElement("hr", null), // Второй дочерний элемент <div/>.
  React.createElement("p", null,   // Третий дочерний элемент <div/>.
                      "This is the sidebar content"));
```

Значение, возвращаемое функцией `React.createElement()`, является обыкновенным объектом JavaScript, который используется фреймворком React для визуализации вывода в окне браузера. Так как раздел посвящен синтаксису JSX, но не React, мы не будем вдаваться в детали возвращаемых объектов `Element` или процесса визуализации. Стоит отметить, что вы можете сконфигурировать инструмент Babel для компиляции элементов JSX в вызовы другой функции, поэтому если вы считаете синтаксис JSX удобным способом выражения вложенных структур данных других видов, то можете задействовать его в собственных сценариях, отличающихся от React.

Важная особенность синтаксиса JSX заключается в том, что вы можете встраивать внутрь выражений JSX обычные выражения JavaScript. Текст в фигурных скобках внутри выражения JSX интерпретируется как простой код JavaScript. Такие вложенные выражения допускается применять в качестве значений атрибутов и как дочерние элементы. Вот пример:

```
function sidebar(className, title, content, drawLine=true) {
  return (
    <div className={className}>
      <h1>{title}</h1>
      { drawLine && <hr/> }
      <p>{content}</p>
    </div>
  );
}
```

Функция `sidebar()` возвращает элемент JSX. Она принимает четыре аргумента, которые использует внутри элемента JSX. Синтаксис с фигурными скобками может напоминать вам шаблонные литералы, в которых применяется `${}` для включения выражений JavaScript внутрь строк. Поскольку мы знаем, что выражения JSX компилируются в вызовы функций, возможность включения произвольных выражений JavaScript не должна вызывать удивление, т.к. вызовы функций также могут записываться с помощью произвольных выражений. Показанный ранее пример кода транслируется инструментом Babel в следующий код:

```
function sidebar(className, title, content, drawLine=true) {
  return React.createElement("div", { className: className },
                        React.createElement("h1", null, title),
                        drawLine && React.createElement("hr", null),
                        React.createElement("p", null, content));
}
```

Код легко читается и воспринимается: фигурные скобки исчезли, а результирующий код естественным образом передает входные параметры функции вызову `React.createElement()`. Обратите внимание на изящный трюк, который мы предприняли здесь с параметром `drawLine` и операцией `&&`, работающей по принципу короткого замыкания. Если вы вызовите `sidebar()` только с тремя аргументами, тогда `drawLine` по умолчанию будет `true` и четвертым аргументом внешнего вызова `createElement()` окажется элемент `<hr/>`. Но если вы передадите `false` в четвертом аргументе функции `sidebar()`, то четвертый аргумент внешнего вызова `createElement()` вычисляется в `false` и элемент

<hr/> не создается. Такое использование операции && является распространенной идиомой в JSX для условного включения или исключения дочернего элемента в зависимости от значения какого-то другого выражения. (Эта идиома работает с React, потому что React просто игнорирует дочерние элементы, которые равны false или null, и не производит для них какой-либо вывод.)

Во время применения выражений JavaScript внутри выражений JSX вы не ограничены простыми значениями наподобие строковых и булевских значений в предыдущем примере. Допускается любое значение JavaScript. На самом деле при программировании с использованием React довольно часто применяются объекты, массивы и функции. Взгляните на следующий пример функции:

```
// Для заданного массива и функции обратного вызова возвращает
// элемент JSX, представляющий списковый HTML-элемент <ul> с массивом
// элементов <li> в качестве своего дочернего элемента.
function list(items, callback) {
  return (
    <ul style={ {padding:10, border:"solid red 4px"} }>
      {items.map((item,index) => {
        <li onClick={() => callback(index)} key={index}>{item}</li>
      })}
    </ul>
  );
}
```

Функция list() использует объектный литерал как значение атрибута style элемента <ul>. (Имейте в виду, что здесь требуются удвоенные фигурные скобки.) Элемент <ul> имеет единственный дочерний элемент, но значением этого дочернего элемента является массив. Дочерний массив — это массив, созданный с применением функции map() на входном массиве для создания массива элементов <li>. (Прием работает с React, потому что фреймворк React разглаживает дочерние элементы элемента, когда визуализирует их. Элемент с одним дочерним элементом типа массива оказывается тем же самым, что и элемент с множеством дочерних элементов, соответствующих элементам массива.) Наконец, обратите внимание, что каждый вложенный элемент <li> имеет атрибут обработчика событий onClick, значением которого является стрелочная функция. Код JSX компилируется в показанный ниже чистый код JavaScript (сформатированный посредством Prettier):

```
function list(items, callback) {
  return React.createElement(
    "ul",
    { style: { padding: 10, border: "solid red 4px" } },
    items.map((item, index) =>
      React.createElement(
        "li",
        { onClick: () => callback(index), key: index },
        item
      )
    )
  );
}
```

Объектные выражения в JSX также используются с операцией распространения (см. подраздел 6.10.4) для указания сразу множества атрибутов. Предположим, что вы пишете много выражений JSX, в которых повторяется общий набор атрибутов. Вы можете упростить свои выражения, определив атрибуты как свойства объекта и "распространить их внутрь" элементов JSX:

```
let hebrew = { lang: "he", dir: "rtl" };  // Указать язык и
                                          // направление письма.
let shalom = <span className="emphasis" {...hebrew}>שלום</span>;
```

Инструмент Babel скомпилирует такой код для применения функции `_extends()` (здесь не показана), которая объединяет атрибут `className` с атрибутами, содержащимися в объекте `hebrew`:

```
let shalom = React.createElement("span",
                                 _extends({className: "emphasis"}, hebrew),
                                 "\u05E9\u05DC\u05D5\u05DD");
```

В заключение отметим еще одну важную особенность JSX, которая пока еще не раскрывалась. Вы видели, что все элементы JSX начинаются с идентификатора сразу после открывающей угловой скобки. Если первая буква этого идентификатора представлена в нижнем регистре (как было во всех примерах), тогда идентификатор передается функции `createElement()` в виде строки. Но если первая буква идентификатора представлена в верхнем регистре, то он трактуется как действительный идентификатор, значение JavaScript которого передается в первом аргументе `createElement()`. Таким образом, выражение JSX вида `<Math/>` компилируется в код JavaScript, который передает функции `React.createElement()` глобальный объект `Math`.

Что касается React, то такая возможность передачи нестроковых значений в качестве первого аргумента функции `createElement()` позволяет создавать *компоненты*. Компонент является способом записи простого выражения JSX (с именем компонента в верхнем регистре), которое представляет более сложное выражение (использующее имена HTML-дескрипторов в нижнем регистре).

Определить новый компонент в React проще всего, написав функцию, которая принимает в своем аргументе "объект свойств" и возвращает выражение JSX. *Объект свойств* — это просто объект JavaScript, который представляет значения атрибутов подобно объектам, передаваемым во втором аргументе функции `createElement()`. Скажем, вот еще один вариант нашей функции `sidebar()`:

```
function Sidebar(props) {
  return (
    <div>
      <h1>{props.title}</h1>
      { props.drawLine && <hr/> }
      <p>{props.content}</p>
    </div>
  );
}
```

Новая функция `Sidebar()` очень похожа на предыдущую функцию `sidebar()`, но она имеет имя, начинающееся с заглавной буквы, и принимает единственный объектный аргумент, а не отдельные аргументы. Это делает функцию `Sidebar()` компонентом React и означает, что ее можно применять вместо имени HTML-дескриптора в выражениях JSX:

```
let sidebar=<Sidebar title="Something snappy" content="Something wise"/>;
```

Такой элемент `<Sidebar/>` компилируется примерно так:

```
let sidebar = React.createElement(Sidebar, {
  title: "Something snappy",
  content: "Something wise"
});
```

Мы написали простое выражение JSX, но когда фреймворк React визуализирует его, он будет передавать второй аргумент (объект `props`) первому аргументу (функции `Sidebar()`) и использовать возвращенное функцией выражение JSX вместо выражения `<Sidebar>`.

17.8. Контроль типов с помощью Flow

Flow (`https://flow.org`) представляет собой языковое расширение, которое позволяет аннотировать код JavaScript информацией о типах, а также инструмент для проверки кода JavaScript (аннотированного и не аннотированного) на предмет ошибок, связанных с типами. Чтобы применять Flow, вы начинаете писать код с использованием языкового расширения Flow для добавления аннотаций типов. Затем вы запускаете инструмент Flow с целью анализа кода и сообщения об ошибках, связанных с типами. После того, как ошибки исправлены и все готово к запуску кода, вы применяете Babel (возможно автоматически как часть процесса пакетирования кода) для удаления из кода аннотаций типов Flow. (Одна из приятных особенностей языкового расширения Flow заключается в том, что нет никакого нового синтаксиса, который инструмент Flow должен был компилировать или трансформировать. Вы используете языковое расширение Flow для добавления аннотаций к коду, а инструменту Babel нужно лишь удалить такие аннотации, возвратив код к стандартному синтаксису JavaScript.)

TypeScript или Flow

TypeScript — очень популярная альтернатива Flow. TypeScript является расширением JavaScript, которое добавляет типы и другие языковые средства. Компилятор TypeScript под названием `tsc` компилирует программы TypeScript в программы JavaScript и в процессе анализирует их, сообщая об ошибках с типами во многом похоже на то, как поступает Flow. Расширение `tsc` — не подключаемый модуль Babel; он является отдельным автономным компилятором.

Простые аннотации типами в TypeScript обычно записываются идентично таким же аннотациям в Flow. При более сложной типизации синтаксис

в двух расширениях отличается, но замысел и ценность двух расширений одинакова. Моя цель в текущем разделе — объяснить преимущества аннотаций типов и статического анализа кода. Я буду делать это через примеры, основанные на Flow, но все, что здесь демонстрируется, также можно обеспечить посредством TypeScript с относительно простыми синтаксическими изменениями.

Расширение TypeScript было выпущено в 2012 году до появления ES6, когда в JavaScript отсутствовали ключевое слово `class`, цикл `for/of`, модули и объекты `Promise`. С другой стороны, Flow — узкое языковое расширение, которое добавляет к JavaScript аннотации типов и ничего больше. Напротив, расширение TypeScript проектировалось в основном как новый язык. Из его названия следует, что главной целью TypeScript было добавление типов к JavaScript и причиной, по которой его применяют в настоящее время. Но типы — не единственное средство, которое TypeScript добавляет к JavaScript: в языке TypeScript есть ключевые слова `enum` и `namespace`, которые в JavaScript попросту отсутствуют. В 2020 году расширение TypeScript лучше интергрировалось с IDE-средами и редакторами кода (в частности с VSCode, который подобно TypeScript произведен Microsoft), нежели Flow.

В конце концов, книга посвящена JavaScript, и я раскрываю здесь Flow, а не TypeScript, поскольку не хочу отвлекаться от JavaScript. Но все, что вы узнаете здесь о добавлении типов к JavaScript, пригодится в случае, если вы решите использовать в своих проектах TypeScript.

Применение Flow требует ответственного отношения, но я обнаружил, что для средних и крупных проектов добавочные усилия стоят того, чтобы их прикладывать. Дополнительное время уходит на добавление в код аннотаций типов, на запуск расширения Flow после каждого редактирования кода и на исправление ошибок, связанных с типами, о которых оно сообщило. Но в ответ Flow будет обеспечивать хорошую дисциплину при написании кода и не позволит идти напролом, что может привести к дефектам. Когда я работал над проектами, где использовалось расширение Flow, то был впечатлен тем количеством ошибок, которые оно находило в моем коде. Возможность устранения проблем до того, как они превратились в дефекты, давало мне великолепное чувство уверенности в том, что мой код корректен.

Когда я впервые начал применять расширение Flow, то обнаружил, что иногда было трудно понять, почему оно жаловалось на мой код. Однако обретя некоторую практику, я пришел к пониманию его сообщений об ошибках и выяснил, что обычно легко внести незначительные изменения в код, чтобы сделать его более безопасным и удовлетворяющим Flow[1]. Я не рекомендую использовать Flow, если вы все еще чувствуете, что изучаете сам язык JavaScript. Но как только вы освоите язык, добавление Flow к вашим проектам JavaScript станет толч-

[1] Если вы программировали на Java, то могли сталкиваться с чем-нибудь подобным при первом написании обобщенного API-интерфейса, использующего параметр типа. Я обнаружил, что процесс изучения Flow удивительно похож на то, что я прошел в 2004 году, когда в Java были добавлены обобщения.

ком к подъему ваших навыков программирования до более высокого уровня. И именно потому я посвящаю последний раздел этой книги краткому руководству по Flow: изучение системы типов JavaScript дает представление о другом уровне или другом стиле программирования.

В настоящем разделе представлено краткое руководство и не предпринимаются попытки полностью охватить расширение Flow. Если вы решили опробовать Flow, тогда почти наверняка потратите время на чтение документации, доступной по ссылке `https://flow.org`. С другой стороны, вам не нужно осваивать систему типов Flow до того, как вы приступите к практическому применению Flow в своих проектах: описанные здесь простые способы использования Flow помогут вам пройти этот долгий путь.

17.8.1. Установка и запуск Flow

Как и другие инструменты, описанные в данной главе, вы можете установить инструмент контроля типов Flow с применением диспетчера пакетов, введя команду вроде `npm install -g flow-bin` или `npm install --save-dev flow-bin`. Если вы установили инструмент глобально с помощью `-g`, тогда можете запускать его посредством команды `flow`. А если вы установили его локально в своем проекте, указав `--save-dev`, то можете запускать его командой `npx flow`. Перед использованием инструмента Flow для выполнения проверки типов сначала запустите его как `flow --init` в корневом каталоге своего проекта, чтобы создать конфигурационный файл `.flowconfig`. Возможно, вам никогда не понадобится добавлять что-то в упомянутый файл, но Flow необходимо знать, где находится корневой каталог вашего проекта.

Когда вы запустите инструмент Flow, он найдет весь исходный код JavaScript в вашем проекте, но будет сообщать об ошибках, связанных с типами, только для файлов, которые "решили участвовать" в контроле типов за счет добавления в свое начало комментария `// @flow`. Такое поведение с согласием участвовать важно, поскольку оно означает, что вы можете задействовать Flow для существующих проектов и начать преобразовывать свой код по одному файлу за раз, не отвлекаясь на сообщения об ошибках и предупреждения, которые относились бы к еще не преобразованным файлам.

Инструмент Flow способен отыскать ошибки в вашем коде, несмотря на то, что вы всего лишь сообщили о желании участвовать с помощью комментария `// @flow`. Даже если вы не применяете языковое расширение Flow и не добавляли аннотаций типов в свой код, инструмент контроля типов Flow по-прежнему может делать заключения о значениях в вашей программе и предупреждать, когда вы используете их несогласованно.

Взгляните на следующее сообщение об ошибке, выданное Flow:

```
Error ------------------------------------ variableReassignment.js:6:3
Ошибка

Cannot assign 1 to i.r because:
Не удается присвоить 1 свойству i.r, потому что:
  • property r is missing in number [1].
  • свойство r отсутствует в числе [1].
```

```
    2| let i = { r: 0, i: 1 };          // Комплексное число 0+1i.
[1] 3| for(i = 0; i < 10; i++) {        // Ой! Переменная цикла
                                        // перезаписывает i.
    4|     console.log(i);
    5| }
    6| i.r = 1;                        // Здесь Flow обнаружит ошибку.
```

В данном случае мы объявляем переменную `i` и присваиваем ей объект. Затем мы применяем `i` снова как переменную цикла, перезаписывая объект. Инструмент Flow замечает это и сигнализирует об ошибке, когда мы пытаемся использовать переменную `i`, как если бы она все еще хранила объект. (Простым исправлением была бы запись начала цикла в виде `for(let i = 0;`, что делает переменную цикла локальной по отношению к циклу.)

Вот еще одна ошибка, которую Flow обнаруживает даже без аннотаций типов:

```
Error                                                    size.js:3:14
Ошибка
Cannot get x.length because property length is missing in number [1].
Не удается получить x.length, потому что свойство length отсутствует
в числе [1].
    1| // @flow
    2| function size(x) {
    3|     return x.length;
    4| }
[1] 5| let s = size(1000);
```

Инструмент Flow видит, что функция `size()` принимает единственный аргумент. Он не знает тип этого аргумента, но способен заметить, что аргумент должен иметь свойство `length`. Когда инструмент Flow выясняет, что функция `size()` вызывается с числовым аргументом, он корректно сигнализирует об ошибке, т.к. числа не имеют свойства `length`.

17.8.2. Использование аннотаций типов

При объявлении переменной JavaScript вы можете добавить к ней аннотацию типа, указав после имени переменной двоеточие и тип:

```
let message: string = "Hello world";
let flag: boolean = false;
let n: number = 42;
```

Инструмент Flow будет знать типы таких переменных, даже если вы их не аннотировали: он способен видеть, какие значения вы присваиваете каждой переменной, и отслеживает это. Тем не менее, если вы добавляете аннотации типов, то Flow знает как тип переменной, так и то, что вы выразили желание о том, чтобы переменная всегда имела указанный тип. Таким образом, если вы применяете аннотацию типа, то Flow будет сигнализировать об ошибке, если вы когда-либо присвоите переменной значение другого типа. Аннотации типов для переменных также особенно полезны, если вы склонны объявлять все свои переменные в начале функции до их использования.

Аннотации типов для аргументов функций похожи на аннотации типов для переменных: после имени аргумента функции указывается двоеточие и имя типа. В случае аннотирования функции вы обычно также добавляете аннотацию для возвращаемого типа функции, что делается между закрывающей круглой скобкой и открывающей фигурной скобкой тела функции. Для функций, которые ничего не возвращают, применяется тип `void` из Flow.

В предыдущем примере мы определили функцию `size()`, которая ожидает аргумент со свойством `length`. Мы можем изменить ее для явного указания, что она ожидает строковый аргумент и возвращает число. Обратите внимание, что теперь Flow сигнализирует об ошибке, если мы передаем функции массив, хотя в этом случае функция работала бы:

```
Error ----------                                    size2.js:5:18
Ошибка

Cannot call size with array literal bound to s because array literal [1]
is incompatible with string [2].
Не удается вызвать size с литералом типа массива, привязанным к s,
потому что литерал типа массива [1] несовместим со строкой [2].

 [2] 2| function size(s: string): number {
     3|   return s.length;
     4| }
 [1] 5| console.log(size([1,2,3]));
```

Аннотации типов также можно использовать со стрелочными функциями, правда, это превращает обычно лаконичный синтаксис в нечто более многословное:

```
const size = (s: string): number => s.length;
```

В отношении Flow важно понимать, что JavaScript-значение `null` имеет Flow-тип `null`, а JavaScript-значение `undefined` — Flow-тип `void`. Но ни одно из таких значений не является членом любого другого типа (если только вы явно не добавите их). Если вы объявляете, что аргумент функции должен быть строкой, тогда он обязан быть строкой, и передача `null` либо `undefined` или пропуск аргумента (что по существу то же самое, что и передача `undefined`) будет считаться ошибкой:

```
Error ----------                                    size3.js:3:18
Ошибка

Cannot call size with null bound to s because null [1] is incompatible
with string [2].
Не удается вызвать size со значением null, привязанным к s,
потому что null [1] несовместимо со строкой [2].

     1| // @flow
 [2] 2| const size = (s: string): number => s.length;
 [1] 3| console.log(size(null));
```

Если вы хотите позволить `null` и `undefined` быть допустимыми значениями для переменной или аргумента функции, тогда просто снабдите имя типа вопросительным знаком. Скажем, применяйте `?string` или `?number` вместо `string`

или number. Если мы изменим нашу функцию size(), чтобы она ожидала аргумент типа ?string, тогда инструмент Flow не будет жаловаться в случае передачи функции значения null. Но теперь он жалуется на кое-что другое:

```
Error ---------- size4.js:3:14
Ошибка

Cannot get s.length because property length is missing in null or
undefined [1].
Не удается получить s.length, потому что свойство length отсутствует
в null или undefined [1].

    1| // @flow
[1] 2| function size(s: ?string): number {
    3|   return s.length;
    4| }
    5| console.log(size(null));
```

Инструмент Flow сообщает нам о том, что записывать s.length небезопасно, поскольку в этом месте нашего кода s может быть null или undefined, а такие значения не имеют свойства length. Именно здесь Flow заботится о том, чтобы мы не шли напролом. Если значение может быть null, то Flow будет настойчиво требовать, чтобы мы проверили такой случай, прежде чем предпринимать действие, которое полагается на то, что значение не должно быть null.

В данной ситуации мы можем устранить проблему, изменив тело функции следующим образом:

```
function size(s: ?string): number {
  // В этом месте кода s может быть строкой либо null или undefined.
  if (s === null || s === undefined) {
    // В этом блоке Flow знает, что s равно null или undefined.
    return -1;
  } else {
    // А в этом блоке Flow знает, что s является строкой.
    return s.length;
  }
}
```

При первом вызове функции аргумент может иметь более одного типа. Но за счет добавления кода для контроля типа мы создаем внутри кода блок, где Flow точно знает, что аргумент является строкой. Когда мы используем s.length внутри этого блока, Flow не жалуется. Обратите внимание, что Flow не требует написания многословного кода наподобие показанного выше. Инструмент Flow также будет удовлетворен, если мы просто заменим тело функции size() кодом return s ? s.length : -1;.

Синтаксис Flow разрешает помещать вопросительный знак перед любой спецификацией типа, чтобы указывать на то, что в дополнение к заданному типу допускаются значения null и undefined. Вопросительный знак также может находиться после имени аргумента, указывая на необязательность самого аргумента. Таким образом, если мы изменим объявление аргумента s из s: ?string на s? : string, то это будет означать, что функцию size() вполне нормально вызывать без аргументов (или со значением undefined, что эквивалентно его

отсутствию), но при ее вызове с аргументом, отличающимся от undefined, аргумент должен быть строкой. В данном случае null — недопустимое значение.

До сих пор мы обсуждали элементарные типы string, number, boolean, null и void и продемонстрировали, как их можно применять с объявлениями переменных, аргументами функций и возвращаемыми значениями функций. В последующих подразделах будут описаны некоторые более сложные типы, поддерживаемые Flow.

17.8.3. Типы классов

Помимо элементарных типов, которые известны инструменту Flow, он также знает обо всех встроенных классах JavaScript и позволяет использовать имена классов как типы. Например, в следующей функции аннотации типов применяются для указания на то, что она должна вызываться с одним объектом Date и одним объектом RegExp:

```
// @flow
// Возвращает true, если представление ISO указанной даты
// соответствует заданному шаблону, и false в противном случае.
// Например:
// const isTodayChristmas = dateMatches(new Date(), /^\d{4}-12-25T/);
export function dateMatches(d: Date, p: RegExp): boolean {
    return p.test(d.toISOString());
}
```

Если вы определяете собственные классы с помощью ключевого слова class, то они автоматически становятся допустимыми типами Flow. Однако чтобы это работало, Flow требует от вас использования аннотаций типов в классе. В частности, каждое свойство класса обязано иметь объявленный тип. Ниже приведен простой класс комплексных чисел, в котором демонстрируется подход:

```
// @flow
export default class Complex {
    // Flow требует расширенного синтаксиса классов, который включает
    // аннотации типов для каждого свойства, используемого классом.
    i: number;
    r: number;
    static i: Complex;
    constructor(r: number, i:number) {
        // Любые свойства, инициализируемые конструктором,
        // обязаны иметь аннотации типов Flow.
        this.r = r;
        this.i = i;
    }
    add(that: Complex) {
        return new Complex(this.r + that.r, this.i + that.i);
    }
}
// Это присваивание не будет разрешено Flow, если внутри
// класса отсутствует аннотация типа для i.
Complex.i = new Complex(0,1);
```

17.8.4. Типы объектов

Тип Flow для описания объекта очень похож на объектный литерал, но только значения свойств заменяются типами свойств. Скажем, вот функция, которая ожидает объект с числовыми свойствами x и y:

```
// @flow
// Для заданного объекта с числовыми свойствами x и y возвращает
// расстояние от начала координат до точки (x,y) в виде числа.
export default function distance(point: {x:number, y:number}): number
{
   return Math.hypot(point.x, point.y);
}
```

В показанном коде текст `{x:number, y:number}` — это тип Flow, в точности как `string` или `Date`. Подобно любому типу вы можете добавлять в начало вопросительный знак для указания на то, что должны быть разрешены также значения `null` и `undefined`.

Внутри объектного типа вы можете дописывать к любому имени свойства вопросительный знак, чтобы указать, что такое свойство является необязательным и может быть опущено. Например, вот как можно записать тип для объекта, который представляет двух- или трехмерную точку:

```
{x: number, y: number, z?: number}
```

Если свойство в объектном типе не помечено как необязательное, тогда оно обязательно и Flow будет сообщать об ошибке, когда соответствующее свойство не присутствует в фактическом значении. Тем не менее, обычно Flow допускает добавочные свойства. Если вы передадите функции `distance()` объект, который имеет свойство `w`, то Flow не пожалуется.

Если вы хотите, чтобы инструмент Flow строго следил за тем, что объект не имеет других свойств кроме явно объявленных в его типе, тогда можете объявить *точный объектный тип*, добавив к фигурным скобкам вертикальные черты:

```
{| x: number, y: number |}
```

Объекты JavaScript временами применяются в качестве словарей или отображений строк на значения. При таком использовании имена свойств заранее не будут известны и не могут объявляться в типе Flow. В случае применения объектов подобным образом вы все равно можете использовать Flow для описания структуры данных. Предположим, что у вас есть объект, свойства которого представляют собой названия крупнейших городов мира, а значения свойств являются объектами, указывающими географическое положение этих городов. Вы могли бы объявить такую структуру данных примерно так:

```
// @flow
const cityLocations :{[string]: {longitude:number, latitude:number}} = {
   "Seattle": { longitude: 47.6062, latitude: -122.3321 },
   // Что делать: добавляйте сюда любые другие важные города.
};
export default cityLocations;
```

17.8.5. Псевдонимы типов

Объекты могут иметь много свойств, а тип Flow, который описывает объект, будет длинным и трудным в наборе. И даже относительно короткие объектные типы могут сбивать с толку, т.к. они очень похожи на объектные литералы. Когда мы выходим за рамки простых типов вроде `number` и `?string`, то часто бывает полезно определять имена для наших типов Flow. Фактически для такой цели в Flow предназначено ключевое слово `type`. Укажите после ключевого слова `type` идентификатор, знак равенства и тип Flow. Затем указанный идентификатор будет служить псевдонимом для типа. Скажем, вот как можно было бы переписать функцию `distance()` из предыдущего подраздела с явно определенным типом `Point`:

```
// @flow
export type Point = {
   x: number,
   y: number
};

// Для заданного объекта Point возвращает его расстояние
// до начала координат.
export default function distance(point: Point): number {
   return Math.hypot(point.x, point.y);
}
```

Обратите внимание, что код импортирует функцию `distance()` и также экспортирует тип `Point`.

Другие модули могут применять `import type Point from './distance.js'`, если хотят использовать такое определение типа. Однако имейте в виду, что `import type` является языковым расширением Flow, а не подлинной директивой импортирования JavaScript. Импорт и экспорт типов применяются инструментом контроля типов Flow, но подобно всем остальным языковым расширениям Flow они удаляются из кода до его запуска.

В заключение стоит отметить, что вместо определения имени для объектного типа Flow, который представляет точку, вероятно, было бы проще и яснее определить класс `Point` и использовать его как тип.

17.8.6. Типы массивов

Тип Flow для описания массива представляет собой составной тип, который также включает тип элементов массива. Например, ниже приведена функция, ожидающая массив чисел, и ошибка, которую Flow сообщает, если вы пытаетесь вызвать эту функцию с массивом, содержащим нечисловые элементы:

```
Error ------ average.js:8:16

Cannot call average with array literal bound to data because string [1]
is incompatible with number [2] in array element.
```

Не удается вызвать average с литералом типа массива, привязанным к data, потому что строка [1] несовместима с числом [2] в элементе массива.

```
[2] 2| function average(data: Array<number>) {
    3|     let sum = 0;
    4|     for(let x of data) sum += x;
    5|     return sum/data.length;
    6| }
    7|
[1] 8| average([1, 2, "three"]);
```

Типом Flow для массива является `Array`, за которым следует тип элементов в угловых скобках. Вы также можете выражать тип массива, указав после типа элементов открывающую и закрывающую квадратные скобки. Таким образом, в нашем примере мы могли бы записать `number[]` вместо `Array<number>`. Я отдаю предпочтение форме записи с угловыми скобками, поскольку существуют другие типы Flow, которые применяют синтаксис с угловыми скобками.

Показанный синтаксис типа `Array` работает для массивов с произвольным количеством элементов, которые все имеют один и тот же тип. Для описания типа *кортежа* в Flow предусмотрен другой синтаксис: массив с фиксированным количеством элементов, каждый из которых может относиться к другому типу. Чтобы выразить тип кортежа, просто запишите типы всех его элементов, отделяя их запятыми, и поместите список в квадратные скобки.

Функция, которая возвращает код состояния и сообщение HTTP, могла бы выглядеть так:

```
function getStatus():[number, string] {
    return [getStatusCode(), getStatusMessage()];
}
```

С функциями, возвращающими кортежи, неудобно работать, если только не использовать деструктурирующее присваивание:

```
let [code, message] = getStatus();
```

Деструктурирующее присваивание плюс возможности назначения псевдонимов типам Flow достаточно облегчают работу с кортежами, так что можете считать их альтернативой классам для простых типов данных:

```
// @flow
export type Color = [number, number, number, number];
                                          // [r, g, b, непрозрачность]
function gray(level: number): Color {
    return [level, level, level, 1];
}
function fade([r,g,b,a]: Color, factor: number): Color {
    return [r, g, b, a/factor];
}
let [r, g, b, a] = fade(gray(75), 3);
```

Теперь, когда у нас есть способ выражения типа массива, давайте возвратимся к определенной ранее функции `size()` и модифицируем ее так, чтобы она ожидала аргумент типа массива, а не строковый аргумент. Мы хотим, чтобы функция имела возможность принимать массив любой длины, поэтому тип

кортежа не подходит. Но было бы нежелательно ограничивать нашу функцию работой только с массивами, в которых все элементы относятся к тому же самому типу. Решением является тип Array<mixed>:

```
// @flow
function size(s: Array<mixed>): number {
   return s.length;
}
console.log(size([1,true,"three"]));
```

Тип элементов mixed указывает на то, что элементы массива могут быть любого типа. Если бы наша функция на самом деле проходила по индексам массива и пыталась работать со всеми его элементами, то Flow настойчиво требовал бы от нас применения проверки с помощью typeof или проверки другого вида для определения типа элемента до выполнения над ним любой небезопасной операции. (Если вы готовы отказаться от контроля типов, то можете также использовать тип any вместо mixed: он позволяет делать со значениями все что угодно, не гарантируя, что они имеют ожидаемый тип.)

17.8.7. Другие параметризированные типы

Вы уже знаете, что при аннотировании значения как Array инструмент Flow требует также указания типа элементов массива внутри угловых скобок. Такой дополнительный тип называется *параметром типа* и Array — не единственный параметризированный класс JavaScript.

Класс Set в JavaScript подобно массиву представляет коллекцию элементов, и вы не можете применять Set как тип сам по себе, но должны включить параметр типа внутри угловых скобок, чтобы указать тип значений, содержащихся внутри множества. (Хотя вы можете использовать mixed или any, если множество будет содержать значения нескольких типов.) Вот пример:

```
// @flow
// Возвращает множество чисел с элементами, которые
// вдвое больше элементов входного множества чисел.
function double(s: Set<number>): Set<number> {
   let doubled: Set<number> = new Set();
   for(let n of s) doubled.add(n * 2);
   return doubled;
}
console.log(double(new Set([1,2,3])));  // Выводится "Set {2, 4, 6}"
```

Еще одним параметризированным типом является Map. Для Map должны быть указаны два параметра типов — тип ключей и тип значений:

```
// @flow
import type { Color } from "./Color.js";

let colorNames: Map<string, Color> = new Map([
   ["red", [1, 0, 0, 1]],
   ["green", [0, 1, 0, 1]],
   ["blue", [0, 0, 1, 1]]
]);
```

Инструмент Flow позволяет вам определять параметры типов также для собственных классов. В приведенном ниже коде определяется класс Result, который параметризируется типами Error и Value. Для представления этих параметров типов в коде мы применяем заполнители E и V. Когда пользователь данного класса объявляет переменную типа Result, он будет указывать фактические типы для подстановки вместо E и V. Объявление переменной может выглядеть так:

```
let result: Result<TypeError, Set<string>>;
```

А вот как определен параметризированный класс:

```
// @flow
// Этот класс представляет результат операции, которая может
// либо сгенерировать ошибку типа E, либо значение типа V.
export class Result<E, V> {
   error: ?E;
   value: ?V;

   constructor(error: ?E, value: ?V) {
      this.error = error;
      this.value = value;
   }

   threw(): ?E { return this.error; }
   returned(): ?V { return this.value; }

   get():V {
      if (this.error) {
         throw this.error;
      } else if (this.value === null || this.value === undefined) {
         throw new TypeError("Error and value must not both be null");
                                 // Ошибка и значение не должны быть null
      } else {
         return this.value;
      }
   }
}
```

Вы даже можете определять параметры типов для функций:

```
// @flow
// Объединяет элементы двух массивов в массив пар.
function zip<A,B>(a:Array<A>, b:Array<B>): Array<[?A,?B]> {
   let result:Array<[?A,?B]> = [];
   let len = Math.max(a.length, b.length);
   for(let i = 0; i < len; i++) {
      result.push([a[i], b[i]]);
   }
   return result;
}

// Создать массив [[1,'a'], [2,'b'], [3,'c'], [4,undefined]].
let pairs: Array<[?number,?string]> = zip([1,2,3,4], ['a','b','c'])
```

17.8.8. Типы, допускающие только чтение

В Flow определено несколько специальных параметризированных "служебных типов", имена которых начинаются с символа $. С большинством таких типов связаны расширенные сценарии использования, которые здесь не рассматриваются. Но два из них довольно полезны на практике. Если вы имеете объектный тип T и хотите создать его версию, допускающую только чтение, тогда просто запишите $ReadOnly<T>. Аналогично с помощью $ReadOnlyArray<T> вы можете описать массив, предназначенный только для чтения, с элементами типа T.

Причина применения таких типов заключается не в том, что они способны предложить гарантию невозможности модификации объекта или массива (см. `Object.freeze()` в разделе 14.2, если вас интересует подлинные объекты, допускающие только чтение), а в том, что они позволяют отлавливать ошибки, связанные с непреднамеренными модификациями. Если вы пишете функцию, которая принимает объект или массив и не изменяет свойства объекта или элементы массива, тогда можете аннотировать аргумент функции одним из типов Flow, предназначенным только для чтения. В таком случае Flow будет сообщать об ошибке, если вы случайно модифицируете входное значение. Ниже показаны два примера:

```
// @flow
type Point = {x:number, y:number};

//Эта функция принимает объект Point, но обязуется не модифицировать его
function distance(p: $ReadOnly<Point>): number {
   return Math.hypot(p.x, p.y);
}

let p: Point = {x:3, y:4};
distance(p)    // => 5

// Эта функция принимает массив чисел, который не будет модифицировать.
function average(data: $ReadOnlyArray<number>): number {
   let sum = 0;
   for(let i = 0; i < data.length; i++) sum += data[i];
   return sum/data.length;
}

let data: Array<number> = [1,2,3,4,5];
average(data)  // => 3
```

17.8.9. Типы функций

Вы уже видели, как добавлять аннотации типов, чтобы указывать типы аргументов функции и ее возвращаемого значения. Но когда один из аргументов функции сам является функцией, нам нужна возможность указания типа такого аргумента-функции.

Чтобы выразить тип функции с помощью Flow, запишите типы каждого аргумента, разделяя их запятыми, поместите список в круглые скобки и затем укажите стрелку с возвращаемым типом функции.

Далее приведен пример функции, которая ожидает, что ей будет передана функция обратного вызова. Обратите внимание на то, как мы определили псевдоним для типа функции обратного вызова:

```
// @flow
// Тип функции обратного вызова, используемой в fetchText().
export type FetchTextCallback = (?Error, ?number, ?string) => void;

export default function fetchText(url: string,
                                  callback: FetchTextCallback) {
   let status = null;
   fetch(url)
      .then(response => {
         status = response.status;
         return response.text()
      })
      .then(body => {
         callback(null, status, body);
      })
      .catch(error => {
         callback(error, status, null);
      });
}
```

17.8.10. Типы объединений

Давайте вернемся еще раз к функции `size()`. На самом деле бессмысленно иметь функцию, которая ничего не делает кроме возвращения длины массива. Для этой цели в массивах предусмотрено свойство `length`. Но функция `size()` может стать полезной, если она будет принимать объект коллекции любого вида (массив, `Set` или `Map`) и возвращать количество элементов в коллекции. В обычном нетипизированном языке JavaScript написать функцию `size()` подобного рода было бы легко. В случае Flow нам необходим способ выражения типа, который допускает массивы, объекты `Set` и объекты `Map`, но не разрешает значения любого другого типа.

Типы такого рода в Flow называются *типами объединений* и позволяют выражать себя путем простого перечисления желаемых типов с разделением их символами вертикальной черты:

```
// @flow
function size(collection: Array<mixed>|Set<mixed>|Map<mixed,mixed>):
number {
   if (Array.isArray(collection)) {
      return collection.length;
   } else {
      return collection.size;
   }
}
size([1,true,"three"]) + size(new Set([true,false]))  // => 5
```

Типы объединений можно читать с использованием слова "или", т.е. "массив или Set или Map", а потому для этого синтаксиса Flow намеренно выбран тот же самый символ вертикальной черты, что и для операции ИЛИ в JavaScript.

Ранее вы видели, что помещение вопросительного знака перед типом разрешает указание значений null и undefined. А теперь выясняется, что префикс ? — это просто сокращенный вариант добавления суффикса |null|void к типу.

В целом, когда вы аннотируете значение типом объединения, Flow не позволит применять такое значение до тех пор, пока вы не сделаете достаточное количество проверок, чтобы выяснить, к какому типу принадлежит фактическое значение. В показанном выше примере функции size() нам необходимо явно проверить, является ли аргумент массивом, прежде чем пытаться получить доступ к свойству length аргумента. Тем не менее, обратите внимание, что мы не обязаны проводить различие между аргументом Set и аргументом Map: в обоих классах определено свойство size и потому код в конструкции else безопасен до тех пор, пока в аргументе не передается массив.

17.8.11. Перечислимые типы и различающие объединения

Flow позволяет использовать элементарные литералы как типы, состоящие из единственного значения. Если вы запишете let x:3;, тогда Flow не разрешит присваивать переменной x никакое другое значение кроме 3. Определять типы, которые имеют только один член, полезно не особенно часто, но объединение литеральных типов может быть полезным. Вероятно, вы сумеете представить себе применение типов следующего вида:

```
type Answer = "yes" | "no";
type Digit = 0|1|2|3|4|5|6|7|8|9;
```

Если вы используете типы, состоящие из литералов, то должны понимать, что будут разрешены только литеральные значения:

```
let a: Answer = "Yes".toLowerCase();     // Ошибка: Answer нельзя
                                         // присваивать строку.
let d: Digit = 3+4;          // Ошибка: Digit нельзя присваивать число.
```

Когда инструмент Flow контролирует ваши типы, в действительности он не производит вычисления, а только проверяет типы вычислений. Инструменту Flow известно, что toLowerCase() возвращает строку, а операция + над числами возвращает число. Хотя мы знаем, что оба вычисления возвращают значения, входящие в тип, Flow не может этого знать и потому сигнализирует об ошибках в обеих строках кода.

Тип объединения литеральных типов вроде Answer и Digit является примером *перечислимого типа*. Канонический сценарий применения перечислимых типов связан с представлением мастей игральных карт:

```
type Suit = "Clubs" | "Diamonds" | "Hearts" | "Spades";
```

Более подходящим примером могут служить коды состояния HTTP:

```
type HTTPStatus =
    | 200      // OK (нормально)
    | 304      // Not Modified (не модифицировано)
    | 403      // Forbidden (запрещено)
    | 404;     // Not Found (не найдено)
```

Один из советов, которые часто слышат начинающие программисты, заключается в том, что следует избегать использования литералов в своем коде, а взамен определять символические константы для представления необходимых значений. Среди причин такого подхода — устранение проблем с опечатками: если вы неправильно запишете строковый литерал наподобие `"Diamonds"`, то интерпретатор JavaScript возможно никогда не пожалуется, но ваш код может работать некорректно. С другой стороны, если вы неправильно наберете идентификатор, тогда интерпретатор JavaScript почти наверняка сгенерирует ошибку, которую вы заметите. В Flow такой совет не всегда применим. Если вы аннотируете переменную типом `Suit` и затем попытаетесь присвоить ей неправильно записанное значение из `Suit`, то Flow будет сигнализировать об ошибке.

Еще один важный сценарий использования литеральных типов — создание *различающих объединений* (discriminated union). При работе с типами объединений (состоящими из фактически разных типов, а не литералов) вы обычно должны писать код для проведения различия между возможными типами. В предыдущем подразделе мы реализовали функцию, которая могла принимать в своем аргумента массив, `Set` или `Map`, и написали код для различения входных данных типа массива от входных данных типа `Set` или `Map`. Если вы хотите создать объединение типов `Object`, то можете облегчить проведение различия между этими типами, применяя литеральный тип внутри каждого индивидуального типа `Object`.

Продемонстрируем такой прием на примере. Предположим, что вы используете в Node поток воркера (см. раздел 16.11) и задействуете метод `postMessage()` и события `"message"` для передачи сообщений на основе объектов между главным потоком и потоком воркера. Существует множество типов событий, которые воркер может посылать главному потоку, но мы хотим создать тип объединения Flow, который опишет все возможные сообщения. Взгляните на следующий код:

```
// @flow
// Воркер посылает сообщение этого типа, когда заканчивает
// покрытие сетчатым узором сплайнов, которые мы ему отправили.
export type ResultMessage = {
   messageType: "result",
   result: Array<ReticulatedSpline>,    // Предполагается, что этот
                                        // тип где-то определен.
};
// Воркер посылает сообщение этого типа, если его код терпит неудачу
// с исключением.
export type ErrorMessage = {
   messageType: "error",
   error: Error,
};
```

```
// Воркер посылает сообщение этого типа для предоставления
// статистических данных по использованию.
export type StatisticsMessage = {
   messageType: "stats",
   splinesReticulated: number,
   splinesPerSecond: number
};
// Когда мы получаем сообщение от воркера,
// оно будет иметь тип WorkerMessage.
export type WorkerMessage =
                  ResultMessage | ErrorMessage | StatisticsMessage;
// Главный поток будет иметь функцию обработчика событий, которой
// передается WorkerMessage. Но поскольку мы осмотрительно определили
// каждый тип сообщения так, что он располагает свойством messageType
// с литеральным типом, обработчик событий может легко проводить
// различие между возможными сообщениями:
function handleMessageFromReticulator(message: WorkerMessage) {
   if (message.messageType === "result") {
      // Только тип ResultMessage имеет свойство messageType
      // с таким значением, поэтому Flow знает, что здесь безопасно
      // использовать message.result.
      // И Flow будет жаловаться, если вы попытаетесь применить
      // любое другое свойство.
      console.log(message.result);
   } else if (message.messageType === "error") {
      // Только тип ErrorMessage имеет свойство messageType
      // со значением "error", поэтому Flow знает, что здесь
      // безопасно использовать message.error.
      throw message.error;
   } else if (message.messageType === "stats") {
      // Только тип StatisticsMessage имеет свойство messageType
      // со значением "stats", поэтому Flow знает, что здесь
      // безопасно применять message.splinesPerSecond.
      console.info(message.splinesPerSecond);
   }
}
```

17.9. Резюме

На сегодняшний день JavaScript является наиболее часто используемым языком в мире. Это живой язык, т.е. язык, продолжающий развиваться и совершенствоваться, который окружен процветающей экосистемой библиотек, инструментов и расширений. В настоящей главе были представлены некоторые из таких инструментов и расширений, но есть еще много чего, о чем следует знать. Экосистема JavaScript процветает, потому что сообщество разработчиков на JavaScript активно, энергично и полно участников, которые делятся своими знаниями через записи в блогах, видеоролики и презентации на конференциях. Когда вы, наконец, закроете эту книгу и присоединитесь к сообществу, то обнаружите обилие источников информации, которые позволят вам быть постоянно в курсе дела и продолжать изучение JavaScript.

С наилучшими пожеланиями Дэвид Флэнаган, март 2020 года.

Предметный указатель

Л

М

П

Содержание