Современный JavaScript для нетерпеливых

Современный JavaScript

для нетерпеливых

Кэй С. Хорстман

ДМК Пресс
Издательство

Кэй С. Хорстман

Современный JavaScript для нетерпеливых

Modern JavaScript for the Impatient

Cay S. Horstmann

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Современный JavaScript для нетерпеливых

Кэй С. Хорстман

Москва, 2021

УДК 004.42Javascript
ББК 32.972
Х79

Хорстман К. С.
Х79 Современный JavaScript для нетерпеливых / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 288 с.: ил.

ISBN 978-5-97060-177-8

Язык JavaScript стремительно набирает популярность: он поддерживается всеми браузерами и все активнее проникает в серверное программирование. Однако даже у опытных программистов, знакомых с такими языками, как Java, C#, C или C++, могут возникнуть затруднения при переходе на JavaScript. Эта книга призвана облегчить их задачу.

В отличие от большинства изданий, посвященных JavaScript, акцент здесь ставится не на переход от прежних версий к современной, а на освоение профессиональными веб-разработчиками нового для них языка программирования. От самых азов автор постепенно переходит к рассмотрению сложных вопросов; темы начального, среднего и высокого уровня помечены в тексте специальными значками.

Изучив книгу, читатель сумеет написать следующую версию своего приложения на современном JavaScript.

УДК 004.42Javascript
ББК 32.972

ISBN 978-0-13-650214-2 (англ.)
ISBN 978-5-97060-177-8 (рус.)

Посвящается Чи – самому важному человеку в моей жизни

Содержание

Предисловие

Опытные программисты, знакомые с такими языками, как Java, C#, C или C++, нередко оказываются в ситуации, когда необходимо поработать с JavaScript. Пользовательские интерфейсы все чаще размещаются в вебе, а JavaScript – язык, поддерживаемый всеми браузерами. Каркас Electron распространяет эту возможность на обогащенные клиентские приложения, и существует несколько решений для создания мобильных JavaScript-приложений. К тому же JavaScript все активнее проникает и в серверное программирование.

Много лет назад JavaScript задумывался как язык для «простенького программирования», набор включенных в него средств приводил в замешательство и мог спровоцировать ошибки в больших программах. Однако предпринятые усилия по стандартизации и созданный инструментарий далеко превзошли первоначальные скромные задумки.

К сожалению, довольно трудно изучить современный JavaScript, не увязнув в трясине старых версий. Целью большинства книг, курсов и статей в блогах является переход от прежних версий JavaScript на современную, что не слишком полезно пришельцам с других языков.

Именно эту проблему призвана решить данная книга. Я предполагаю, что читатель – знающий программист, который понимает, что такое ветвления, циклы, функции, структуры данных, и знаком с основами объектно-ориентированного программирования. Я объясню, что значит быть продуктивным программистом на современном JavaScript, лишь в скобках упоминая об ушедших в прошлое средствах. Вы узнаете, как поставить себе на службу современный JavaScript, избежав древних ловчих ям.

JavaScript, быть может, и не идеален, но, как показывает практика, хорошо приспособлен для программирования пользовательских интерфейсов и многих серверных задач. Как прозорливо заметил Джефф Этвуд, «любое приложение, которое *можно* написать на JavaScript, в конце концов *будет* написано на JavaScript».

Проработав эту книгу, вы сумеете написать следующую версию своего приложения на современном JavaScript!

Пять золотых правил

Держась подальше от немногих «классических» средств JavaScript, вы сможете заметно упростить себе освоение и использование языка. Возможно, прямо сейчас эти правила покажутся вам бессмысленными, но я все же приведу их, чтобы ссылаться в дальнейшем. И не пугайтесь – их совсем немного.

1. При объявлении переменных употребляйте ключевые слова `let` или `const`, а не `var`.
2. Пользуйтесь строгим режимом.

3. Обращайте внимание на типы и избегайте автоматического преобразования типов.
4. Разберитесь, что такое прототипы, но для работы с классами, конструкторами и методами применяйте современный синтаксис.
5. Не используйте ключевое слово `this` вне конструкторов и методов.

И еще одно метаправило: *избегайте «что это?!»* – фрагментов странного JavaScript-кода, сопровождаемых саркастическим «Что это?!». Некоторым доставляет удовольствие демонстрировать якобы ужасы JavaScript, анатомируя запутанный код. Я ни разу не почерпнул ничего полезного, спускаясь в эту кроличью нору. Зачем, к примеру, знать, что `2 * ['21']` равно 42, а `2 + ['40']` не равно, если золотое правило 3 призывает избегать преобразований типов? В общем случае, оказываясь в сбивающей с толку ситуации, я задаю себе вопрос, как избежать ее, а не как объяснить ее таинственные, но бесполезные детали.

Пути к познанию

Работая над книгой, я старался помещать информацию туда, где вы сможете найти ее, когда понадобится. Но это необязательно самое подходящее место при первом прочтении книги. Чтобы помочь вам проложить собственный путь к познанию, я пометил каждую главу значком, обозначающим ее базовый уровень сложности. Разделы же, более сложные, чем глава в целом, помечены собственными значками. Вы можете без опаски пропускать такие разделы и возвращаться к ним, когда будете готовы воспринять материал.

Вот эти значки.

Нетерпеливый кролик означает тему **начального** уровня, пропускать которую не должен даже самый нетерпеливый читатель.

Алиса обозначает тему среднего уровня, с которой стоило бы познакомиться большинству программистов, но, возможно, не при первом чтении.

Чеширский кот обозначает тему **повышенного** уровня, от которой расплывется в улыбке лицо разработчика каркасов. Большинство прикладных программистов могут спокойно пропустить эти разделы.

Наконец, значок Безумного шляпника сопровождает **сложную**, способную свести с ума тему, предназначенную только для тех, кто одержим нездоровым любопытством.

Краткое содержание книги

В главе 1 рассказывается об основных понятиях JavaScript: значениях и их типах, переменных и, самое важное, объектных литералах. В главе 2 описы-

вается поток управления. Если вы знакомы с Java, C# или C++, можете пролистать ее по диагонали. В главе 3 вы узнаете о функциях и функциональном программировании – вещи, крайне важной в JavaScript. Объектная модель в JavaScript сильно отличается от языков программирования, основанных на классах. Глава 4 посвящена деталям с упором на современный синтаксис. В главах 5 и 6 описаны библиотечные классы, которые чаще всего используются при работе с числами, датами, строками и регулярными выражениями. В основном это материал начального уровня, но встречаются разделы повышенного типа.

Следующие четыре главы посвящены темам промежуточного уровня. В главе 7 вы научитесь работать с массивами и другими коллекциями, имеющимися в стандартной библиотеке JavaScript. Если ваша программа рассчитана на пользователей со всего мира, то обратите особое внимание на вопросы интернационализации, которые освещаются в главе 8. Глава 9 об асинхронном программировании чрезвычайно важна для всех программистов. Асинхронное программирование на JavaScript когда-то считалось весьма сложным предметом, но после включения в язык обещаний и ключевых слов `async` и `await` значительно упростилось. Теперь в JavaScript имеется стандартная система модулей, которая описывается в главе 10. Вы узнаете, как использовать модули, написанные другими программистами, и как создать свой собственный.

В главе 11 рассматривается метапрограммирование на повышенном уровне. Читать ее имеет смысл, если вы собираетесь создать инструмент для анализа и преобразования произвольных JavaScript-объектов. В главе 12 описание JavaScript завершается рассмотрением еще одной продвинутой темы: итераторов и генераторов – мощных механизмов, предназначенных для организации обхода коллекций и порождения произвольных последовательностей значений.

Наконец, имеется дополнительная глава 13, посвященная TypeScript. TypeScript – это надмножество JavaScript, добавляющее проверку типов на этапе компиляции. Не будучи частью стандартного JavaScript, эта надстройка очень популярна. Прочитайте эту главу и сами решите, к чему склоняетесь: к обычному JavaScript или к системе типов на этапе компиляции.

Цель данной книги – заложить прочные основы для уверенного использования самого *языка* JavaScript. За информацией о постоянно изменяющихся инструментах и каркасах придется обратиться в другое место.

Почему я написал эту книгу

JavaScript – один из самых широко распространенных языков программирования на планете. Как и многие программисты, я был знаком с *ломаным* JavaScript, однако настал день, когда нужно было спешно научиться писать на JavaScript по-серьезному. Но как?

Есть немало учебников по основам JavaScript для непрофессиональных веб-разработчиков, но на таком уровне я его и так знал. *Книга с носорогом*

Флэнагана[1] была чудом в 1996 году, но теперь нагружает читателей слишком большим наследием прошлого. Книга Крокфорда «JavaScript: The Good Parts»[2] стала сигналом к действию в 2008-м, но многие содержащиеся в ней призывы уже вошли в последующие версии языка. Существует множество книг, помогающих программистам на JavaScript старой школы вступить в мир новых стандартов, но в них предполагается хорошее знакомство с «классическим» JavaScript, чем я похвастаться не мог.

Разумеется, веб кишит блогами на тему JavaScript разного качества – одни содержат точную и систематическую информацию, другие – просто случайный набор фактов. На мой взгляд, просеивать веб-блоги и оценивать степень их достоверности – занятие не слишком эффективное. Как ни странно, я не смог найти ни одной книги для миллионов программистов, которые знают Java или другой подобный язык и хотят изучить JavaScript в его современном виде, не обремененном историческим багажом.

Поэтому мне пришлось написать ее самому.

Благодарности

Я хотел бы еще раз поблагодарить своего редактора Грега Денча (Greg Doench) за поддержку этого проекта, а также Дмитрия и Алину Кирсановых за корректуру и верстку книги. Отдельное спасибо рецензентам Гэйлу Андерсону (Gail Anderson), Тому Остину (Tom Austin), Скотту Дэвису (Scott Davis), Скотту Гуду (Scott Good), Кито Манну (Kito Mann), Бобу Николсону (Bob Nicholson), Рону Маку (Ron Mak) и Генри Тремблею (Henri Tremblay), которые исправно указали на ошибки и внесли продуманные предложения по улучшению книги.

Кэй Хорстманн
Берлин
Март 2020

[1] *David Flanagan.* JavaScript: The Definitive Guide. Sixth Edition (O'Reilly Media, 2011).

[2] Вышла в издательстве O'Reilly Media в 2008 году.

Об авторе

Кэй С. Хорстманн – главный автор книг «Core Java™», т. I и II, 11-е изд. (Pearson, 2018), «Scala for the Impatient», 2-е изд. (Addison-Wesley, 2016) и «Core Java SE 9 for the Impatient» (Addison-Wesley, 2017). Кэй – заслуженный профессор информатики в университете Сан-Хосе, пропагандист Java, часто выступает на конференциях по компьютерной тематике.

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Springer очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Значения и переменные

В этой главе вы узнаете о типах данных в JavaScript-программе: числах, строках и других примитивных типах, а также об объектах и массивах. Вы увидите, как сохранять значения в переменных, как преобразовывать значения из одного типа в другой и как применять к значениям операторы для получения новых значений.

Даже самые отчаянные программисты на JavaScript согласятся, что некоторые языковые конструкции – задуманные для того, чтобы сократить размер программ, – могут давать интуитивно неочевидные результаты, поэтому их лучше избегать. В этой и следующих главах я буду обращать внимание на такие проблемы и сформулирую простые правила безопасного программирования.

1.1. Запуск JavaScript

Выполнять встречающиеся в этой книге программы можно несколькими способами.

Изначально задумывалось, что JavaScript будет выполняться внутри браузера. Можно встроить JavaScript-код в HTML-файл и, вызвав метод `window.alert`, отобразить значения. Вот пример такого файла:

```
<html>
  <head>
    <title>My First JavaScript Program</title>
    <script type="text/javascript">
      let a = 6
      let b = 7
      window.alert(a * b)
    </script>
  </head>
  <body>
  </body>
</html>
```

Просто откройте этот файл в своем любимом браузере – и в диалоговом окне увидите результат (см. рис. 1.1).

Рис. 1.1 ❖ Выполнение JavaScript-кода в браузере

Можно набрать короткую последовательность команд на консоли, которая является частью комплекта средства разработки, входящего в состав браузера. Чтобы открыть средства разработки, нажмите соответствующую клавишу или выберите пункт из меню (во многих браузерах это клавиша **F12** или комбинация **Ctrl**+**Alt**+**I**, а в Mac – **Cmd**+**Alt**+**I**). Затем перейдите на вкладку **Console** и введите свой JavaScript-код (рис. 1.2).

Рис. 1.2 ❖ Выполнение JavaScript-кода на консоли разработчика

Третий способ – установить Node.js с сайта http://nodejs.org. Затем откройте терминал и запустите программу `node`, которая входит в цикл «читать-выполнять-печатать» (цикл REPL). Вводите команды и смотрите на их результаты (рис. 1.3).

Рис. 1.3 ❖ Выполнение JavaScript-кода на консоли разработчика

Если последовательность команд длиннее, поместите ее в файл и вызовите метод `console.log`, чтобы увидеть результат. Например, поместите следующие команды в файл `first.js`:

```
let a = 6
let b = 7
console.log(a * b)
```

Затем выполните команду

```
node first.js
```

Результат команды `console.log` будет выведен в окно терминала.

Можно также воспользоваться средой разработки, например Visual Studio Code, Eclipse, Komodo или WebStorm. Все они позволяют редактировать и выполнять JavaScript-код, как показано на рис. 1.4.

1.2. Типы и оператор typeof

Значения в JavaScript могут иметь следующие типы:

- число;
- булево значение `false` или `true`;
- специальные значения `null` и `undefined`;
- строка;
- символ;
- объект.

Все типы, кроме объекта, собирательно называются *примитивными*.

Подробнее об этих типах рассказано в следующих разделах, за исключением символов, о которых речь пойдет в главе 11.

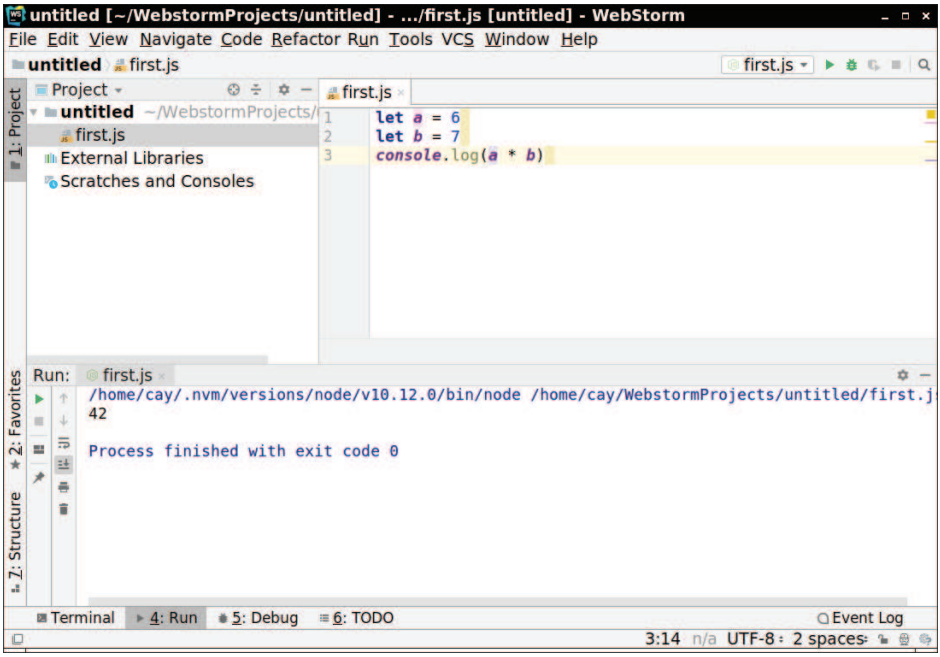

Рис. 1.4 ❖ Выполнение JavaScript-кода в среде разработки

Чтобы узнать тип значения, следует воспользоваться оператором `typeof`, который возвращает одну из строк `'number'`, `'boolean'`, `'undefined'`, `'object'`, `'string'`, `'symbol'` и еще нескольких. Например, вызов `typeof 42` возвращает строку `'number'`.

> **Примечание.** Хотя тип `null` отличается от типа `object`, значением `typeof null` является строка `'object'`. Так сложилось исторически.

> **Предостережение.** Как и в Java, можно сконструировать объекты, обертывающие числа, булевы значения и строки. Например, `typeof new Number(42)` и `typeof new String('Hello')` возвращают `'object'`. Однако в JavaScript нет причин конструировать такие обертки. Поскольку подобные действия могут приводить к недоразумениям, стандарты кодирования нередко их явно запрещают.

1.3. Комментарии

В JavaScript есть два вида комментариев. Однострочный комментарий начинается двумя литерами // и продолжается до конца строки, например:

```
// как-то так
```

Комментарии, заключенные между парами литер /* и */, могут занимать несколько строк, например:

```
/*
  как-то
  так
*/
```

В этой книге комментарии набраны моноширинным шрифтом для простоты восприятия. Понятно, что в текстовом редакторе они, скорее всего, будут выделены цветом.

> **Примечание.** В отличие от Java, в JavaScript нет специальных комментариев для оформления документации. Однако существуют сторонние инструменты, например JSDoc (http://usejsdoc.org), предлагающие аналогичную функциональность.

1.4. Объявления переменных

Для сохранения значения в переменной служит предложение `let`:

```
let counter = 0
```

В JavaScript у переменных нет типа. В любой переменной можно сохранить значение любого типа. Например, допустимо заменить содержимое `counter` строкой:

```
counter = 'zero'
```

Почти никогда так делать не стоит. Но бывают ситуации, когда наличие нетипизированных переменных упрощает написание обобщенного кода, работающего с разными типами.

Если переменная явно не инициализирована, то она принимает специальное значение `undefined`:

```
let x // объявляет x и присваивает ей значение undefined
```

> **Примечание.** Вы, наверное, обратили внимание, что предложения не завершаются точкой с запятой. В JavaScript, как и в Python, точка с запятой в конце строки необязательна. В Python даже считается неподобающим добавлять ненужные точки с запятой. Но программисты на JavaScript по этому вопросу разделились на два лагеря. Мы обсудим аргументы за и против в главе 2. Вообще-то, я стараюсь не занимать ничью сторону в пустопорожних спорах, но в этой книге мне пришлось выбрать что-то одно. Я остановился на стиле «без точки с запятой» по одной простой причине: чтобы код не был похож на Java или C++. Глядя на фрагмент кода, можно сразу сказать, что он написан на JavaScript.

Если значение переменной не планируется изменять, то следует объявить ее в предложении `const`:

```
const PI = 3.141592653589793
```

Попытка модифицировать так объявленное значение приведет к ошибке во время выполнения.

В одном предложении const или let можно объявить несколько переменных:

```
const FREEZING = 0, BOILING = 100
let x, y
```

Но многие программисты предпочитают объявлять каждую переменную в отдельной строке.

> **Предостережение.** Избегайте двух устаревших форм объявления переменных: с помощью ключевого слова var и вообще без ключевого слова:
>
> ```
> var counter = 0 // устаревшая форма
> coutner = 1 // обратите внимание на опечатку – будет создана новая переменная!
> ```
>
> У объявления с помощью var есть несколько серьезных недостатков, о них будет сказано в главе 3. Создание при первом присваивании, очевидно, опасно. Если сделать опечатку в имени переменной, то будет создана новая переменная. По этой причине такое поведение считается ошибкой в *строгом режиме*, запрещающем устаревшие конструкции. В главе 3 я расскажу, как включить строгий режим.
>
> **Совет.** В предисловии я перечислил пять золотых правил, следование которым позволит устранить большую часть недоразумений, вызванных «классическими» средствами JavaScript. Первые два из них гласят:
>
> 1. При объявлении переменных употребляйте ключевые слова let или const, а не var.
> 2. Пользуйтесь строгим режимом.

1.5. Идентификаторы

Имя переменной должно быть выбрано с соблюдением общего синтаксиса *идентификаторов*. Идентификатор может включать буквы Юникода, цифры и литеры _ и $. Первая литера не должна быть цифрой. Имена, включающие литеру $, иногда используются в библиотеках и инструментальных средствах. Некоторые программисты применяют идентификаторы, начинающиеся или заканчивающиеся знаком подчеркивания, чтобы показать, что речь идет о «закрытых» членах. В своих именах лучше избегать использования $, а также _ в начале и в конце. Подчерки внутри имени не вызывают никаких нареканий, но многие JavaScript-программисты предпочитают «верблюжью нотацию» camelCase, когда границы слов обозначаются сменой регистра.

Следующие ключевые слова не разрешается использовать в качестве идентификаторов:

```
break case catch class const continue debugger default delete do
else enum export extends false finally for function if import in instanceof
new null return super switch this throw true try typeof var void while with
```

В строгом режиме запрещены также такие ключевые слова:

```
implements interface let package protected private public static
```

Следующие ключевые слова добавлены в язык недавно; их можно использовать в качестве идентификаторов ради обратной совместимости, но лучше этого не делать:

```
await as async from get of set target yield
```

> **Примечание.** В идентификаторах разрешено использовать любые буквы и цифры Юникода, например:
>
> `const π = 3.141592653589793`
>
> Однако это не принято, потому что у многих программистов нет метода ввода таких литер.

1.6. Числа

В JavaScript нет явного типа целого числа. Все числа с плавающей точкой двойной точности. Конечно, можно использовать и целые значения, просто не обращайте внимания на разницу между `1` и `1.0` (к примеру). А как насчет округления? Целые числа в диапазоне от `Number.MIN_SAFE_INTEGER` ($-2^{53} + 1$, или `-9 007 199 254 740 991`) и `Number.MAX_SAFE_INTEGER` ($2^{53} - 1$, или `9 007 199 254 740 991`) представляются точно. Диапазон целых чисел шире, чем в Java. Коль скоро результат остается в этом же диапазоне, арифметические операции над целыми также точны. Но при выходе за границы диапазона возникают ошибки округления. Например, вычисление `Number.MAX_SAFE_INTEGER * 10` дает `90071992547409900`.

> **Примечание.** Если диапазона целых чисел недостаточно, можно воспользоваться «большими целыми», число цифр в которых не ограничено. Большие целые описываются в главе 5.

Как и в любом языке программирования, избежать ошибок округления при операциях над числами с плавающей точкой невозможно. Например, `0.1 + 0.2` дает `0.30000000000000004` – точно так же, как в Java, C++ или Python. Это неизбежно, поскольку десятичные числа вроде 0.1, 0.2 или 0.3 не имеют точного двоичного представления. Для вычислений с долларами и центами следует представлять все денежные суммы в центах.

Другие формы числовых литералов, в частности шестнадцатеричные числа, описаны в главе 5.

Для преобразования строки в число предназначены функции `parseFloat` и `parseInt`:

```
const notQuitePi = parseFloat('3.14') // число 3.14
const evenLessPi = parseInt('3') // целое число 3
```

Метод `toString` преобразует число обратно в строку:

```
const notQuitePiString = notQuitePi.toString() // строка '3.14'
const evenLessPiString = (3).toString() // строка '3'
```

Примечание. В JavaScript, как и в C++, но не как в Java, имеются функции и методы. Функции `parseFloat` и `parseInt` не являются методами, поэтому для их вызова не нужна точка.

Примечание. Как видно из предыдущего фрагмента, методы можно вызывать от имени числовых литералов. Однако при этом нужно заключить литерал в круглые скобки, чтобы точка не интерпретировалась как десятичный разделитель.

Предостережение. Что, если использовать дробное число там, где ожидается целое? Все зависит от ситуации. Допустим, мы хотим выделить подстроку. Тогда дробная часть числа позиций отбрасывается:

```
'Hello'.substring(0, 2.5) // строка 'He'
```

Но если указать дробный индекс, то результат будет равен `undefined`:

```
'Hello'[2.5] // undefined
```

Не стоит тратить время на то, чтобы выяснить, когда дробное число можно использовать вместо целого. Оказавшись в такой ситуации, проясните свое намерение, вызвав функцию `Math.trunc(x)`, чтобы отбросить дробную часть, или `Math.round(x)`, чтобы округлить до ближайшего целого.

Результатом деления на ноль является `Infinity` или `-Infinity`. Однако `0 / 0` равно `NaN` – константе, обозначающей «не число».

Некоторые функции, порождающие целые числа, возвращают `NaN`, когда на вход передано недопустимое значение. Например, `parseFloat('pie')` равно `NaN`.

1.7. Арифметические операторы

В JavaScript имеются обычные операторы `+ - * /` сложения, вычитания, умножения и деления. Заметим, что оператор `/` всегда возвращает результат с плавающей точкой, даже если оба операнда целые. Например, `1 / 2` равно `0.5`, а не 0, как было бы в Java или C++.

Оператор `%` для неотрицательных целых операндов возвращает остаток от целочисленного деления, как в Java, C++ и Python. Например, если `k` – неотрицательное целое число, то `k % 2` равно `0`, если `k` четное, и `1`, если `k` нечетное.

Если `k` и `n` – положительные значения, быть может дробные, то `k % n` равно значению, которое получается повторным вычитанием `n` из `k`, до тех пор пока результат не станет меньше `n`. Например, `3.5 % 1.2` равно `1.1` – результату вычитания `1.2` дважды. По поводу отрицательных операндов см. упражнение 3.

Оператор `**` обозначает «возведение в степень», как в Python (и восходит еще к языку Fortran). `2 ** 10` равно `1024`, `2 ** -1` равно `0.5`, в `2 ** 0.5` равно квадратному корню из 2.

Если операндом любого арифметического оператора является «не число» `NaN`, то и результатом является `NaN`.

Как и в Java, C++ и Python, арифметические операции можно совмещать с присваиванием:

```
counter += 10 // то же, что counter = counter + 10
```

Операторы инкремента ++ и декремента -- увеличивают и уменьшают переменную на единицу:

```
counter++ // то же, что counter = counter + 1
```

> **Предостережение.** Как Java и C++, JavaScript следует по пути языка C, в котором оператор ++ может находиться до или после переменной. Это дает фактически две операции: прединкремента и постикремента.
>
> ```
> let counter = 0
> let riddle = counter++
> let enigma = ++counter
> ```
>
> Каковы значения `riddle` и `enigma`? Если не знаете, можете догадаться, тщательно проанализировав приведенное выше описание, или просто попробовать, или почерпнуть мудрости из интернета. Но лично я настоятельно рекомендую никогда не писать код, зависящий от этого знания.
>
> Некоторые программисты считают операторы ++ и -- настолько предосудительными, что вообще никогда их не используют. Да и необходимости особой в них нет – в конце концов, `counter += 1` ненамного длиннее, чем `counter++`. В этой книге я буду пользоваться операторами ++ и --, но только не в ситуации, когда их значение захватывается.

Как и в Java, оператор + применяется также для конкатенации строк. Если `s` – строка, а `x` – значение любого типа, то `s + x` и `x + s` – строки, полученные преобразованием `x` в строку и дописыванием результата в конец `s`.

Например:

```
let counter = 7
let agent = '00' + counter // строка '007'
```

> **Предостережение.** Как видим, выражение `x + y` является числом, если оба операнда – числа, и строкой, если хотя бы один операнд – строка. Во всех остальных случаях правила сложнее, а результаты не столь полезны. Либо оба операнда преобразуются в строки и конкатенируются, либо оба преобразуются в числа и складываются. Например, выражение `null + undefined` вычисляется как числовое сложение `0 + NaN`, которое дает `NaN` (см. табл. 1.1).
>
> Для остальных арифметических операторов делается только попытка преобразовать операнды в числа. Например, значением `6 * '7'` является `42` – строка `'7'` преобразуется в число 7.

Таблица 1.1 Преобразование в числа и строки

Значение	В число	В строку
Число	Само число	Строка, состоящая из цифр числа
Строка, состоящая из цифр числа	Значение числа	Сама строка
Пустая строка `''`	0	`''`
Любая другая строка	`NaN`	Сама строка
`false`	`0`	`'false'`
`true`	`1`	`'true'`
`null`	`0`	`'null'`
`undefined`	`NaN`	`'undefined'`

Таблица 1.1 *(окончание)*

Значение	В число	В строку
Пустой массив `[]`	`0`	`''`
Массив, содержащий одно число	Это число	Строка, состоящая из цифр числа
Прочие массивы	`NaN`	Элементы массива преобразуются в строки и разделяются запятыми, например `'1,2,3'`
Объекты	По умолчанию `NaN`, но это поведение можно настроить	По умолчанию `'[object Object]'`, но это поведение можно настроить

Совет. Не полагайтесь на автоматическое преобразование типов в арифметических операциях. Правила запутанные, а результат может оказаться неожиданным. Если хотите обрабатывать операнды, являющиеся строками или одноэлементными массивами, преобразуйте их явно.

Совет. Отдавайте предпочтение шаблонным литералам (см. раздел 1.11 «Шаблонные литералы»), а не конкатенации строк. Тогда не нужно будет помнить, как оператор + воздействует на нечисловые операнды.

1.8. Булевы значения

У булева типа имеется два значения: `false` и `true`. В условии значения любого типа преобразуются в булевы. Значения `0`, `NaN`, `null`, `undefined` и пустые строки преобразуются в `false`, все остальные – в `true`.

Кажется, что все просто, но, как мы увидим в следующей главе, результаты могут сбить с толку кого угодно. Чтобы не стать жертвой недоразумения, рекомендуется использовать в условиях только настоящие булевы значения.

1.9. null и undefined

В JavaScript есть два способа обозначить отсутствие значения. Если переменная объявлена, но не инициализирована, то ее значением будет `undefined`. Обычно такое случается в функциях. Если при вызове функции не указан некоторый параметр, то вместо него подставляется `undefined`.

Значение `null` служит для того, чтобы обозначить намеренное отсутствие значения.

Полезно ли это различие? По этому вопросу мнения разделились. Некоторые программисты считают, что наличие двух значений «ничего» чревато ошибками, и предлагают использовать только одно. В таком случае следует остановиться на `undefined`. Избежать использования `undefined` в языке JavaScript невозможно, но почти всегда можно обойтись без `null`.

Сторонники противоположной точки зрения полагают, что никогда не следует присваивать переменной значение `undefined` или возвращать `undefined`

из функции, а вместо отсутствующих значений всегда нужно употреблять `null`. В таком случае `undefined` может означать наличие серьезной проблемы.

> **Совет.** В любом проекте явно выберите тот или другой подход: используйте либо `undefined`, либо `null`, чтобы явно обозначить отсутствие значения. В противном случае вы ввяжетесь в бессмысленные философские дискуссии и будете без нужды проверять и на `undefined`, и на `null`.

> **Предостережение.** В отличие от `null`, слово `undefined` *не* является зарезервированным. Это переменная в глобальной области видимости. Когда-то давным-давно глобальной переменной `undefined` можно было присвоить новое значение! Очевидно, что эта идея была чудовищной, так что теперь `undefined` – константа. Однако по-прежнему можно объявлять *локальные* переменные с именем `undefined`. Но, конечно, эта идея ничем не лучше. Также не стоит объявлять локальные переменные `NaN` и `Infinity`.

1.10. Строковые литералы

Строковые литералы заключаются в одиночные или двойные кавычки: `'Hello'` или `"Hello"`. В этой книге всюду используются одиночные кавычки.

Если внутри строки используется та же кавычка, что ограничивающие ее, экранируйте кавычку обратной косой чертой. Также следует экранировать саму обратную черту и управляющие символы (см. табл. 1.2).

Например, `'\\\'\'\\\n'` – строка длины 5, содержащая литеры `\''\`, за которыми следует знак новой строки.

Таблица 1.2. Управляющие последовательности для специальных символов

Управляющая последовательность	Название	Значение в Юникоде
`\b`	Забой	`\u{0008}`
`\t`	Табуляция	`\u{0009}`
`\n`	Перевод строки	`\u{000A}`
`\r`	Возврат каретки	`\u{000D}`
`\f`	Прогон страницы	`\u{000C}`
`\v`	Вертикальная табуляция	`\u{000B}`
`\'`	Одиночная кавычка	`\u{0027}`
`\"`	Двойная кавычка	`\u{0022}`
`\\`	Обратная косая черта	`\u{005C}`
`\`новая строка	Переход на следующую строку	Ничего – символ новой строки не добавляется: «`Hel\` `lo`» равно строке «`Hello`»

Чтобы включить в строку JavaScript произвольные символы Юникода, нужно просто набрать или вставить их, при условии что для файла задана соответствующая кодировка (например, UTF-8):

```
let greeting = 'Hello 🌐'
```

Если важно, чтобы файлы хранились в кодировке ASCII, то можно воспользоваться нотацией `\u{`*кодовая точка*`}`:

```
let greeting = 'Hello \u{1F310}'
```

К сожалению, JavaScript отличается неприятной особенностью в отношении Юникода. Чтобы понять, в чем ее суть, нужно обратиться к истории. До появления Юникода существовали несовместимые кодировки символов, т. е. одна и та же последовательность байтов могла означать совершенно разные вещи для читателей из США, России или Китая.

Юникод и проектировался для решения этих проблем. Когда в 1980-х годах унификация только начиналась, казалось, что 16-битового кода будет достаточно, чтобы охватить все языки мира, да еще и место для будущих расширений останется. В 1991 году вышла версия Unicode 1.0, в которой было занято чуть меньше половины имеющихся 65 536 кодовых точек. Когда в 1995 году вышли JavaScript и Java, оба языка уже поддерживали Юникод, и строки в них были представлены последовательностями 16-битовых значений.

Но, конечно, со временем произошло неизбежное – количество символов в Юникоде перевалило за 65 536. Сейчас используется 21 бит, и все уверены, что уж этого-то точно хватит. Однако JavaScript так и застрял на 16-битовых значениях.

Чтобы объяснить, как эта проблема решена, нам понадобится ввести терминологию. *Кодовой точкой* Юникода называется 21-битовое значение, с которым ассоциирован символ. В JavaScript используется кодировка UTF-16, в которой все кодовые точки Юникода представлены одним или двумя 16-битовыми значениями, которые называются *кодовыми единицами*. Для символов до `\u{FFFF}` нужна одна кодовая единица. Остальные символы кодируются двумя единицами, которые берутся из зарезервированной области, где нет закодированных символов. Например, `\u{1F310}` кодируется как последовательность `0xD83C 0xDF10`. (Описание алгоритма кодирования см. по адресу http://en.wikipedia.org/wiki/UTF-16.)

Детали кодирования вам знать необязательно, но нужно понимать, что для одних символов требуется одна 16-битовая кодовая единица, а для других две.

Например, «длина» строки `'Hello 🌐'` равна 8, хотя она содержит семь символов Юникода (обратите внимание на пробел между `Hello` и 🌐). Для доступа к кодовым единицам строки можно использовать оператор `[]`. Выражение `greeting[0]` представляет собой строку, состоящую из одной буквы `'H'`. Но оператор `[]` не работает с символами, для кодирования которых нужно две кодовые единицы. Кодовые единицы символа 🌐 занимают позиции 6 и 7. Выражения `greeting[6]` и `greeting[7]` – строки длины 1, каждая из которых занимает одну кодовую единицу, которым не соответствует никакой символ. Иными словами, это недопустимые строки Юникода.

> **Совет.** В главе 2 мы узнаем, как перебрать отдельные кодовые точки в строке с помощью цикла `for`.

Примечание. 16-битовые кодовые единицы можно задать в строковом литерале. В этом случае нужно опустить фигурные скобки: `\uD83C\uDF10`. Для кодовых единиц не больше `\u{0xFF}` можно воспользоваться «шестнадцатеричной управляющей последовательностью», например `\xA0` вместо `\u{00A0}`. Но я не вижу разумных причин ни для того, ни для другого.

В главе 6 мы узнаем о различных методах для работы со строками.

Примечание. В JavaScript имеются также литералы для регулярных выражений – см. главу 6.

1.11. Шаблонные литералы

Шаблонным литералом называется строка, которая может содержать выражения и занимать несколько строчек. Такие строки заключаются в обратные кавычки (`` `...` ``), например:

```
let destination = 'world' // обычная строка
let greeting = `Hello, ${destination.toUpperCase()}!` // шаблонный литерал
```

Выражения внутри `${...}` вычисляются, при необходимости преобразуются в строку и подставляются в шаблон. В данном случае результатом будет строка

```
Hello, WORLD!
```

Шаблонные литералы могут быть вложенными, т. е. одна конструкция `${...}` может встречаться внутри другой:

```
greeting = `Hello, ${firstname.length > 0 ? `${firstname[0]}. ` : '' } ${lastname}`
```

Все знаки новой строки внутри шаблонного литерала включаются в строку. Например, в результате вычисления

```
greeting = `<div>Hello</div>
<div>${destination}</div>
`
```

переменной `greeting` присваивается строка `'<div>Hello</div>\n<div>World</div>\n'`, в которой каждая строчка завершается знаком перевода строки. (Принятые в Windows окончания строчек `\r\n`, встречающиеся в исходном файле, перед вставкой в строку преобразуются в окончания `\n`, принятые в Unix.)

Чтобы включить в строку знаки обратной кавычки, доллара или обратной косой черты, следует экранировать их знаком обратной косой черты: `` `\`\$\\` `` – строка, содержащая три символа: `` `$\ ``.

Примечание. *Тегированным шаблонным литералом* называется шаблонный литерал, которому предшествует функция, например:

```
html`<div>Hello, ${destination}</div>`
```

Здесь вызывается функция `html`, которой передаются фрагменты шаблона `'<div>Hello, '` и `'</div>'`, и значение выражения `destination`.
В главе 6 мы узнаем, как писать собственные теговые функции.

1.12. Объекты

Объекты в JavaScript сильно отличаются от тех, которые встречаются в основанных на классах языках типа Java и C++. В JavaScript объект – это просто совокупность пар имя-значение, или «свойств», например:

```
{ name: 'Harry Smith', age: 42 }
```

У такого объекта есть только открытые данные, ни о какой инкапсуляции или поведении и речи не идет. Объект не является экземпляром какого-то класса. Иными словами, он не имеет ничего общего с тем, что традиционно понимается под объектом в объектно-ориентированном программировании. В главе 4 мы увидим, что объявлять классы и методы можно, но совсем не так, как в большинстве других языков.

Разумеется, объект можно сохранить в переменной:

```
const harry = { name: 'Harry Smith', age: 42 }
```

Имея такую переменную, можно обращаться к свойствам объекта с помощью стандартной нотации с точкой:

```
let harrysAge = harry.age
```

Можно модифицировать существующие свойства и добавлять новые:

```
harry.age = 40
harry.salary = 90000
```

> **Примечание.** Переменная `harry` была объявлена как `const`, но, как мы только что видели, объект, на который она ссылается, можно изменять. Однако присвоить новое значение `const`-переменной невозможно:
>
> ```
> const sally = { name: 'Sally Lee' }
> sally.age = 28 // OK - изменился объект, на который ссылается sally
> sally = { name: 'Sally Albright' }
>   // Ошибка - нельзя присвоить новое значение const-переменной
> ```
>
> Иными словами, `const` ведет себя как `final` в Java и совсем не так, как `const` в C++.

Для удаления свойства служит оператор `delete`:

```
delete harry.salary
```

Попытка доступа к несуществующему свойству дает `undefined`:

```
let boss = harry.supervisor // undefined
```

Имя свойства может быть результатом вычисления. Тогда для доступа к значению свойства нужно использовать квадратные скобки:

```
let field = 'Age'
let harrysAge = harry[field.toLowerCase()]
```

1.13. Синтаксис объектного литерала

Это первый раздел промежуточного уровня в данной главе. Если вы только приступаете к изучению JavaScript, можете пропускать разделы, помеченные таким значком, без ущерба для понимания.

Объектный литерал может завершаться запятой. Это упрощает добавление новых свойств по мере эволюции кода:

```
let harry = {
  name: 'Harry Smith',
  age: 42, // дальше могут быть добавлены новые свойства
}
```

Часто при объявлении объектного литерала значения свойств хранятся в переменных, имена которых совпадают с именами свойств. Например:

```
let age = 43
let harry = { name: 'Harry Smith', age: age }
  // Свойству 'age' присвоено значение переменной age
```

Для такой ситуации предусмотрен сокращенный синтаксис:

```
let harry = { name: 'Harry Smith', age } // теперь свойство age равно 43
```

Для задания вычисляемых имен свойств в объектных литералах употребляются квадратные скобки:

```
let harry = { name: 'Harry Smith', [field.toLowerCase()] : 42 }
```

Именем свойства может быть только строка. Если имя не удовлетворяет правилам формирования идентификатора, заключите его в кавычки:

```
let harry = { name: 'Harry Smith', 'favorite beer': 'IPA' }
```

Для доступа к таким свойствам нельзя использовать нотацию с точкой. Применяйте квадратные скобки:

```
harry['favorite beer'] = 'Lager'
```

Такие имена свойств встречаются нечасто, но иногда бывают удобны. Например, именами свойств объекта могут быть имена файлов, а значениями – содержимое этих файлов.

> **Предостережение.** Иногда при синтаксическом разборе встречаются ситуации, когда открывающая фигурная скобка может быть началом объектного литерала или блока. В таких случаях предпочтение отдается блоку. Например, если ввести на консоли браузера или в Node.js выражение
>
> ```
> {} - 1
> ```
>
> то будет выполнен пустой блок, а затем вычислено и отображено выражение –1. С другой стороны, в выражении
>
> ```
> 1 - {}
> ```

{} – пустой объект, который преобразуется в NaN. После этого отображается результат вычисления (тоже NaN).

На практике такая неоднозначность обычно не возникает. Созданный объектный литерал, как правило, сохраняется в переменной, которая передается в качестве аргумента или возвращается в качестве результата. В этих ситуациях синтаксический анализатор не ожидает блока.

Если вы столкнетесь с ситуацией, в которой объектный литерал ошибочно принят за блок, лекарство простое: заключите литерал в круглые скобки. Пример будет приведен в разделе 1.16 «Деструктуризация».

1.14. Массивы

В JavaScript массив – это просто объект, в котором именами свойств являются строки `'0'`, `'1'`, `'2'` и т. д. (Строки используются, потому что числа не могут быть именами свойств.)

Для объявления литералов массива нужно заключить их элементы в квадратные скобки:

```
const numbers = [1, 2, 3, 'many']
```

Это объект, имеющий пять свойств: `'0'`, `'1'`, `'2'`, `'3'` и `'length'`.

Свойство `length` на единицу больше самого большого индекса, преобразованного в число. Значением `numbers.length` является число `4`.

Доступ к первым четырем свойствам осуществляется с помощью квадратных скобок: `numbers['1']` равно `2`. Для удобства аргумент внутри скобок автоматически преобразуется в строку. Можно писать также `numbers[1]`, что создает иллюзию, будто мы работаем с таким же массивом, как в Java или C++.

Заметим, что типы элементов в массиве не обязательно должны быть одинаковыми. Массив `numbers` содержит три числа и строку.

Некоторые элементы массива могут отсутствовать:

```
const someNumbers = [ , 2, , 9] // отсутствуют свойства '0', '2'
```

Как и в любом объекте, несуществующее свойство принимает значение `undefined`. Например, `someNumbers[0]` и `someNumbers[6]` равны `undefined`.

Новые элементы можно добавлять за концом массива:

```
someNumbers[6] = 11 // теперь длина someNumbers равна 7
```

Как и для любого объекта, свойства массива, на который ссылается `const`-переменная, можно изменять.

Примечание. *Завершающая* запятая *не* означает, что элемент отсутствует. Например, в массиве `[1, 2, 7, 9,]` четыре элемента, и индекс последнего равен 3. Как и в случае объектных литералов, завершающие запятые ставятся на случай, если литерал может расширяться со временем, например:

```
const developers = [
  'Harry Smith',
  'Sally Lee',
  // добавляйте новые элементы над этой строкой
]
```

Поскольку массив – это объект, можно добавлять в него произвольные свойства:

```
numbers.lucky = true
```

Это необычный, но вполне допустимый в JavaScript прием.

Оператор `typeof` для массива возвращает строку `'object'`. Чтобы проверить, является ли объект массивом, нужно вызвать метод `Array.isArray(obj)`.

Когда массив необходимо преобразовать в строку, все его элементы преобразуются в строки и соединяются запятыми. Например, выражение

```
'' + [1, 2, 3]
```

равно строке `'1,2,3'`.

Массив длины 0 преобразуется в пустую строку.

В JavaScript, как и в Java, нет многомерных массивов, но их можно имитировать с помощью массива массивов, например:

```
const melancholyMagicSquare = [
  [16, 3, 2, 13],
  [5, 10, 11, 8],
  [9, 6, 7, 12],
  [4, 15, 14, 1]
]
```

Теперь для доступа к элементу нужно использовать две пары квадратных скобок:

```
melancholyMagicSquare[1][2] // 11
```

В главе 2 мы увидим, как обойти все элементы массива. За полным обсуждением всех методов массива обратитесь к главе 7.

1.15. JSON

JavaScript Object Notation, или JSON, – это облегченный текстовый формат для обмена данными объектов между приложениями (необязательно написанными на JavaScript).

В двух словах, в JSON используется синтаксис JavaScript для объектных и массивовых литералов с несколькими ограничениями:

- значениями могут быть объектные литералы, массивовые литералы, строки, числа с плавающей точкой, а также `true`, `false` и `null`;
- все строки заключаются в двойные, а не в одиночные кавычки;
- все имена свойств заключаются в двойные кавычки;
- не допускаются завершающие запятые и пропущенные элементы.

Формальное описание нотации см. на сайте `www.json.org`.

Приведем пример JSON-строки:

```
{ "name": "Harry Smith", "age": 42, "lucky numbers": [17, 29], "lucky": false }
```

Метод `JSON.stringify` преобразует объект JavaScript в JSON-строку, а метод `JSON.parse` разбирает JSON-строку, возвращая объект JavaScript. Эти методы часто используются при взаимодействии с сервером по протоколу HTTP.

> **Предостережение.** Метод `JSON.stringify` пропускает свойства объектов, имеющие значение `undefined`, а элементы массива, равные `undefined`, преобразует в `null`. Например, значением `JSON.stringify({ name: ['Harry', undefined, 'Smith'], age: undefined })` является строка `'{"name":["Harry",null,"Smith"]}'`.

Некоторые программисты используют метод `JSON.stringify` для протоколирования. Команда

```
console.log(`harry=${harry}`)
```

выводит бесполезное сообщение

```
harry=[object Object]
```

Исправит ситуацию команда `JSON.stringify`:

```
console.log(`harry=${JSON.stringify(harry)}`)
```

Заметим, что эта проблема возникает, только когда строка содержит объекты. Если протоколировать объект сам по себе, то на консоли он отображается правильно. Простая альтернатива – протоколировать имена и значения по отдельности:

```
console.log('harry=', harry, 'sally=', sally)
```

Или еще проще – сделать их частями объекта:

```
console.log({harry, sally}) // протоколируеся объект { harry: { ... }, sally: { ... } }
```

1.16. Деструктуризация

Деструктуризация – это удобный синтаксис для выборки элементов массива или значений объекта. Как и все остальные темы промежуточного уровня, можете пропустить ее и вернуться, когда будете готовы.

В этом разделе мы опишем базовый синтаксис, а в следующем рассмотрим нюансы.

Сначала разберем массивы. Пусть имеется массив `pair` с двумя элементами. Конечно, можно получить элементы следующим образом:

```
let first = pair[0]
let second = pair[1]
```

Или применить деструктуризацию:

```
let [first, second] = pair
```

В этом предложении переменные `first` и `second` объявлены и инициализированы элементами `pair[0]` и `pair[1]`.

Левая часть деструктурирующего присваивания на самом деле не является массивовым литералом, ведь переменные `first` и `second` еще не существуют. Считайте, что левая часть – это образец, описывающий, как следует сопоставлять переменные с правой частью.

Рассмотрим более сложный случай – сопоставление переменных с элементами массива:

```
let [first, [second, third]] = [1, [2, 3]]
  // first получает значение 1, second - 2, third - 3
```

Массив в правой части может быть длиннее образца в левой части. Ни с чем не сопоставленные элементы просто игнорируются:

```
let [first, second] = [1, 2, 3]
```

Если же массив короче, то несопоставленным переменным присваивается значение `undefined`:

```
let [first, second] = [1]
  // first получает значение 1, second - undefined
```

Если переменные `first` и `second` уже объявлены, то деструктуризацию можно использовать для присваивания им новых значений:

```
[first, second] = [4, 5]
```

> **Совет.** Чтобы обменять значения переменных x и y, достаточно написать:
>
> ```
> [x, y] = [y, x]
> ```

Когда деструктуризация используется для присваивания, левая часть необязательно должна состоять только из переменных. Можно использовать произвольные *l-значения*, т. е. выражения, которые могут встречаться в правой части оператора присваивания. Так, следующее предложение – допустимая деструктуризация:

```
[numbers[0], harry.age] = [13, 42] // то же, что numbers[0] = 13; harry.age = 42
```

Объекты деструктурируются аналогично, только вместо позиций в массиве нужно использовать имена свойств:

```
let harry = { name: 'Harry', age: 42 }
let { name: harrysName, age: harrysAge } = harry
```

В этом примере две переменные `harrysName` и `harrysAge` объявляются и инициализируются значениями свойств `name` и `age` объекта в правой части.

Помните, что левая часть *не* является объектным литералом. Это образец, показывающий, как переменные сопоставляются с правой частью.

Деструктуризация объекта полезнее, когда имя свойства совпадает с именем переменной. В этом случае имя свойства и запятую можно опустить. В следующем предложении переменные `name` и `age` объявляются и инициализируются одноименными свойствами объекта в правой части:

```
let { name, age } = harry
```

Это то же самое, что

```
let { name: name, age: age } = harry
```

или

```
let name = harry.name
let age = harry.age
```

> **Предостережение.** Если деструктуризация объекта используется для присваивания значений уже существующим переменным, то выражение присваивания следует заключить в скобки:
>
> ```
> ({name, age} = sally)
> ```
>
> В противном случае синтаксический анализатор сочтет открывающую фигурную скобку началом блока.

1.17. Еще о деструктуризации

В предыдущем разделе я показал только самые простые и практически полезные части синтаксиса деструктуризации. А сейчас мы познакомимся с более мощными и интуитивно не столь очевидными возможностями. Можете пропустить этот раздел и вернуться к нему, когда освоите основы.

1.17.1. Дополнительные сведения о деструктуризации объектов

Можно деструктурировать вложенные объекты:

```
let pat = { name: 'Pat', birthday: { day: 14, month: 3, year: 2000 } }
let { birthday: { year: patsBirthYear } } = pat
  // Переменная patsBirthYear объявляется и инициализируется значением 2000
```

Еще раз отметим, что левая часть второго предложения – *не объект*, а образец для сопоставления переменных с правой частью. Это предложение эквивалентно такому:

```
let patsBirthYear = pat.birthday.year
```

Как и в случае объектных литералов, поддерживаются вычисляемые имена свойств:

```
let field = 'Age'
let { [field.toLowerCase()]: harrysAge } = harry
  // Присваивается значение harry[field.toLowerCase()]
```

1.17.2. Объявление прочих

При деструктуризации массива можно записать все оставшиеся элементы в массив. Для этого нужно добавить префикс ... перед именем переменной:

```
numbers = [1, 7, 2, 9]
let [first, second, ...others] = numbers
  // first получает значение 1, second - 7, others - [2, 9]
```

Если в массиве в правой части недостаточно элементов, то переменная для хранения прочих будет равна пустому массиву:

```
let [first, second, ...others] = [42]
  // first получает значение 42, second - undefined, others - []
```

Объявление прочих работает и для объектов:

```
let { name, ...allButName } = harry
  // allButName равно { age: 42 }
```

Переменной `allButName` присваивается объект, содержащий все свойства, кроме `name`.

1.17.3. Значения по умолчанию

Для каждой переменной можно задать значение по умолчанию, которое будет использоваться, если искомого значения нет в объекте или в массиве, или если есть, но равно `undefined`. Поставьте знак `=` и выражение после имени переменной:

```
let [first, second = 0] = [42]
  // first получает значение 42, second - 0, потому что в правой части нет подходящего
  // элемента
let { nickname = 'None' } = harry
  // nickname получает значение 'None', потому что у harry нет свойства nickname
```

В выражениях по умолчанию можно использовать переменные, которым уже присвоены значения:

```
let { name, nickname = name } = harry
  // name и nickname получают значение harry.name
```

Ниже приведено типичное применение деструктуризации со значениями по умолчанию. Пусть имеется объект, описывающий детали некоторой обработки, например инструкции форматирования. Если какое-то свойство не задано, то мы хотим использовать значение по умолчанию:

```
let config = { separator: '; ' }
const { separator = ',', leftDelimiter = '[', rightDelimiter = ']' } = config
```

Здесь переменная `separator` инициализирована символом-разделителем, а ограничители по умолчанию используются, потому что не заданы в конфи-

гурации. Синтаксис деструктуризации значительно лаконичнее, чем искать каждое свойство, проверять, определено ли оно, и подставлять значение по умолчанию, если не определено.

В главе 3 мы встретимся с похожим случаем, когда деструктуризация используется для формирования параметров функции.

Упражнения

1. Что будет, если прибавить `0` к значениям `NaN`, `Infinity`, `false`, `true`, `null` и `undefined`? Что будет, если конкатенировать пустую строку с `NaN`, `Infinity`, `false`, `true`, `null` и `undefined`? Сначала сообразите, а потом проверьте свою догадку.
2. Чему равны выражения `[] + []`, `{} + []`, `[] + {}`, `{} + {}`, `[] - {}`? Сравните результаты их вычисления в командной строке и присваивания переменной. Объясните то, что видите.
3. В Java и в C++ (в отличие от Python, который в этом отношении следует столетиям математического опыта) `n % 2` равно `-1`, если `n` – отрицательное целое число. Исследуйте поведение оператора `%` для отрицательных операндов. Проанализируйте как целые числа, так и числа с плавающей точкой.
4. Пусть `angle` – некоторый угол, выраженный в градусах, который после прибавления или вычитания других углов может принимать произвольные значения. Вы хотите нормализовать его, так чтобы он попадал в диапазон от 0 (включая) до 360 (не включая). Как это сделать с помощью оператора `%`?
5. Придумайте как можно больше способов породить строку с двумя знаками обратной косой строки `\\` в JavaScript, пользуясь описанными в этой главе механизмами.
6. Придумайте как можно больше способов породить строку из одного символа 🌐 в JavaScript.
7. Приведите реалистичный пример шаблонной строки с вложенным выражением, которое содержит еще одну шаблонную строку с вложенным выражением.
8. Предложите три способа создать массив с «дыркой» в последовательности индексов.
9. Объявите массив с элементами в позициях `0`, `0.5`, `1`, `1.5` и `2`.
10. Что происходит, когда массив массивов преобразуется в строку?
11. Создайте два объекта, представляющих людей, и сохраните их в переменных `harry` и `sally`. В каждый объект включите свойство `friends`, которое содержит массив друзей. Предположим, что `harry` – друг `sally`, а `sally` – друг `harry`. Что произойдет при вызове метода `log` для каждого объекта? А если вызвать метод `JSON.stringify`?

Глава 2

Управляющие конструкции

В этой главе вы узнаете об управляющих конструкциях в языке JavaScript: ветвлениях, циклах и перехвате исключений. Также приводится обзор предложений JavaScript и описывается процесс автоматической вставки точки с запятой.

2.1. Выражения и предложения

В JavaScript, как и в Java и C++, выражения и предложения различаются. Выражение имеет значение. Например, `6 * 7` – выражение, его значение равно `42`. Вызов метода, например `Math.max(6, 7)`, – еще один пример выражения.

У предложения нет значения. Оно выполняется, и в результате достигается какой-то эффект. Например,

```
let number = 6 * 7;
```

– предложение. Его эффект – объявление и инициализация переменной `number`. Такое предложение называется объявлением переменной.

Помимо объявлений переменных, есть еще два распространенных типа предложений: ветвления и циклы. Мы встретимся с ними в этой главе чуть ниже.

Простейший вид предложения – *предложение выражения*. Оно состоит из выражения, за которым следует точка с запятой, например:

```
console.log(6 * 7);
```

Выражение `console.log(6 * 7)` имеет побочный эффект – отобразить `42` на консоли. У него также имеется значение, равное `undefined`, поскольку метод `console.log` не возвращает чего-то поинтереснее. Даже если бы у выражения было более интересное значение, оно ничего бы нам не дало, поскольку значение предложения выражения отбрасывается.

Таким образом, предложение выражения полезно, только если у выражения имеется побочный эффект. Предложение выражения

```
6 * 7;
```

– допустимый JavaScript-код, но программе от него никакого проку.

Понимать разницу между предложениями и выражениями полезно, но в JavaScript непросто увидеть, чем выражение отличается от предложения выражения. В следующем разделе мы узнаем, что если строчка содержит единственное выражение, то в конец автоматически добавляется запятая, превращая его в предложение. Поэтому невозможно наблюдать выражение на JavaScript-консоли браузера или в Node.js.

Например, попробуйте набрать `6 * 7`. Будет выведено значение выражения:

```
6 * 7
42
```

В этом и заключается задача цикла читать-вычислять-печатать (REPL): он читает выражение, вычисляет его и печатает значение.

Вот только из-за автоматической вставки точки с запятой цикл REPL в JavaScript REPL на самом деле видит *предложение*

```
6 * 7;
```

У предложений нет значений, но REPL в JavaScript все равно отображает значения.

Попробуйте ввести объявление переменной:

```
let number = 6 * 7;
undefined
```

Как мы только что видели, цикл REPL отображает значение выражения. В случае объявления переменной цикл REPL отображает `undefined`. В упражнении 1 мы выясним, что отображается для других предложений.

Экспериментируя с циклом REPL, важно понимать, как интерпретировать результаты. Например, введите следующее предложение выражения и посмотрите на ответ:

```
console.log(6 * 7);
42
undefined
```

Первая строка результата – побочный эффект вызова `console.log`. Вторая строка – значение, возвращенное этим вызовом. Как уже было сказано, метод `console.log` возвращает `undefined`.

2.2. Вставка точки с запятой

В JavaScript некоторые предложения должны заканчиваться точкой с запятой. Среди прочих к ним относятся объявления переменных, предложения выражения и предложения нелинейной передачи управления (`break`, `continue`, `return`, `throw`). Однако JavaScript услужливо вставляет точки с запятой за вас.

Основное правило простое. Во время обработки предложения синтаксический анализатор включает все лексемы, пока не встретит точку с запятой или «постороннюю лексему» – нечто такое, что не может быть частью предложения. Если посторонней лексеме предшествует знак завершения строки или закрывающая скобка } либо встретился конец данных, то анализатор добавляет точку с запятой.

Например:

```
let a = x
  + someComplicatedFunctionCall()
let b = y
```

После первой строки точка с запятой не добавляется. Лексема + в начале второй строки не является «посторонней».

Но лексема let в начале третьей строки посторонняя. Она не может быть частью объявления первой переменной. Поскольку посторонняя лексема встретилась после знака завершения строки, вставляется точка с запятой:

```
let a = x
  + someComplicatedFunctionCall();
let b = y
```

Правило «посторонней лексемы» простое и пригодно почти во всех случаях. Однако оно не срабатывает, если предложение *начинается* лексемой, которая могла бы быть частью предыдущего предложения. Рассмотрим пример:

```
let x = a
(console.log(6 * 7))
```

После a точка с запятой не вставляется.

Синтаксически

```
a(console.log(6 * 7))
```

– корректный JavaScript-код: вызывается функция a, которой передается значение, возвращенное в результате обращения к console.log. Иными словами, лексема (во второй строке *не* посторонняя.

Конечно, это искусственный пример. Скобки вокруг console.log(6 * 7) были необязательны. А вот другой пример, который приводят довольно часто:

```
let a = x
[1, 2, 3].forEach(console.log)
```

Поскольку [может встретиться после x, точка с запятой не вставляется. В том маловероятном случае, если вы захотите обойти таким способом массивовый литерал, сохраните массив в переменной:

```
let a = x
const numbers = [1, 2, 3]
numbers.forEach(console.log)
```

Совет. Никогда не начинайте предложение литерами (или [. Тогда не нужно будет думать о том, что это предложение может интерпретироваться как продолжение предыдущей строки.

Примечание. Если точка с запятой отсутствует, то строка, начинающаяся шаблонным литералом или литеральным регулярным выражением, может быть объединена с предыдущей строкой, например:

```
let a = x
`Fred`.toUpperCase()
```

Здесь x`Fred` разбирается как тегированный шаблонный литерал. Но на практике вы вряд ли станете писать такой код. При работе со строкой или регулярным выражением мы хотим использовать результат, поэтому литерал не окажется в начале предложения.

Второе правило точки с запятой более проблематично. Точка с запятой вставляется после предложения нелинейной передачи управления (`break`, `continue`, `return`, `throw`, `yield`), за которым сразу следует знак завершения строки. Если написать

```
return
  x + someComplicatedExpression;
```

то точка с запятой будет добавлена автоматически:

```
return ;
  x + someComplicatedExpression;
```

Функция возвращает управление, не возвращая значения. Вторая строка – предложение выражения, которое никогда не выполняется.

Исправление тривиально. Не ставьте знак новой строки сразу после `return`. Поместите хотя бы одну лексему возвращаемого выражения на той же строке:

```
return x +
  someComplicatedExpression;
```

Помнить об этом правиле нужно, даже если вы всегда и всюду расставляете точки с запятой самостоятельно.

Помимо правил «посторонней лексемы» и «нелинейной передачи управления», есть еще одно малоизвестное правило. Точка с запятой вставляется, если последовательности литер `++` или `--` непосредственно предшествует знак завершения строки.

Согласно этому правилу,

```
x
++
y
```

означает

```
x;
++y;
```

Если вы всегда будете размещать ++ на той же строке, что и операнд, то об этом правиле можно не думать.

Эти правила автоматической вставки – часть языка. На практике они вполне терпимы. Если вам нравятся точки с запятой, пожалуйста, ставьте их. Не нравятся – опускайте. Но *в любом случае* нужно помнить о нескольких потаенных уголках.

> **Примечание.** Точки с запятой вставляются *только* перед знаком завершения строки или закрывающей скобкой }. Если в одной строке расположено несколько предложений, то точки с запятой нужно ставить явно:
>
> ```
> if (i < j) { i++; j-- }
> ```
>
> Здесь точка с запятой необходима, чтобы разделить предложения `i++` и `j--`.

2.3. Ветвления

Если вы знакомы с любым из языков C, C++, Java или C#, то можете спокойно пропустить этот раздел.

Условное предложение в JavaScript имеет вид

```
if (условие) предложение
```

Условие должно находиться внутри круглых скобок.

> **Совет.** Лучше, чтобы результатом вычисления условия было `true` или `false`, пусть даже JavaScript допускает произвольные значения и преобразует их в булевы. В следующем разделе мы увидим, что такие преобразования не всегда интуитивно очевидны и потенциально опасны. Следуйте золотому правилу 3 из предисловия:
>
> - обращайте внимание на типы и избегайте автоматического преобразования типов.

Часто требуется выполнить несколько предложений, если условие удовлетворяется. В таком случае пользуйтесь *предложением блока*, которое имеет вид:

```
{
  statement1
  statement2
  ...
}
```

Необязательная ветвь `else` выполняется, если условие не удовлетворяется, например:

```
if (yourSales > target) {
  performance = 'Good'
  bonus = 100
} else {
  performance = 'Mediocre'
  bonus = 0
}
```

> **Примечание.** В этом примере демонстрируется «единственно правильный стиль расстановки фигурных скобок» (one true brace style), когда открывающая фигурная скобка ставится в конце строки, предшествующей первому предложению блока. Этот стиль часто используется в программах на JavaScript.

Если ветвь `else` содержит еще одно предложение `if`, то, по соглашению, используется такой формат:

```
if (yourSales > 2 * target) {
  performance = 'Excellent'
  bonus = 1000
} else if (yourSales > target) {
  performance = 'Good'
  bonus = 100
} else {
  performance = 'Mediocre'
  bonus = 0
}
```

Фигурные скобки вокруг одиночного предложения необязательны:

```
if (yourSales > target)
  bonus = 100
```

> **Предостережение.** Если в предложении `if/else` не использовать фигурные скобки или использовать, но не в «единственно правильном стиле расстановки фигурных скобок», то можно получить код, который будет работать, когда программа запускается из файла, но перестанет, будучи скопированным на консоль JavaScript. Рассмотрим пример:
>
> ```
> if (yourSales > target)
>   bonus = 100
> else
>   bonus = 0
> ```
>
> Некоторые консоли JavaScript анализируют код построчно. Такая консоль будет думать, что предложение `if` закончилось до ветви `else`. Чтобы не сталкиваться с этой проблемой, используйте скобки или размещайте все предложение `if` в одной строчке.
>
> ```
> if (yourSales > target) bonus = 100; else bonus = 0
> ```

Иногда удобно иметь выражение, аналогичное предложению `if`. Рассмотрим вычисление большего из двух чисел:

```
let max = undefined
if (x > y) max = x; else max = y
```

Было бы красивее инициализировать `max` большим из чисел `x` и `y`. Поскольку `if` – предложение, мы не можем написать:

```
let max = if (x > y) x else y // ошибка - неожиданное предложение if
```

Вместо этого воспользуйтесь «условным» оператором `? :`. Результатом вычисления выражения *условие ? первое : второе* является *первое*, если *ус-*

ловие удовлетворяется, и *второе* в противном случае. Тем самым наша задача решена:

```
let max = x > y ? x : y
```

> **Примечание.** Выражение `x > y ? x : y` – удобный пример для иллюстрации условного оператора, но если вам нужно найти большее из двух чисел, то лучше пользоваться функцией `Math.max` из стандартной библиотеки.

2.4. Булевость

В этом разделе, отмеченном значком Безумного шляпника, описывается одна сбивающая с толку особенность JavaScript. Если вы готовы следовать приведенной в предыдущем разделе рекомендации – употреблять в условиях только булевы значения, – то можете пропустить данный раздел.

В JavaScript условие (в частности, в предложении `if`) необязательно должно принимать булево значение. Если условие принимает «похоже на false» значение – `0`, `NaN`, `null`, `undefined` или пустую строку, – то считается, что оно не выполнено. Все остальные значения «похожи на true», для них условие выполняется. Термины «похожий на false» и «похожий на true» в спецификации языка не встречаются.

> **Примечание.** Понятие «булевости» применимо к условиям цикла for, операндам логических операторов `&&`, `||` и `!`, а также к первому операнду оператора `? :`. Все эти конструкции рассматриваются ниже в этой главе.

Правило преобразования к типу Boolean на первый взгляд выглядит логично. Предположим, что имеется переменная `performance`, и нам нужно только узнать, принимает ли она значение `undefined`. Тогда можно написать:

```
if (performance) ... // опасно
```

Конечно, эта проверка не пройдет, если `performance` равно `undefined`. Не пройдет она и тогда, когда `performance` равна `null`.

Но что, если `performance` – пустая строка? Или число 0? Действительно ли мы хотим трактовать эти значения как отсутствие значения? Иногда да, а иногда нет. Быть может, лучше явно выразить свое намерение и написать:

```
if (performance !== undefined) ...
```

2.5. Сравнение

В JavaScript имеется обычный набор операторов сравнения:

- `<` меньше;
- `<=` меньше или равно;
- `>` больше;
- `>=` больше или равно.

Когда сравниваются числа, эти операторы не таят в себе никаких неожиданностей:

```
3 < 4 // true
3 >= 4 // false
```

Любое сравнение с `NaN` дает `false`:

```
NaN < 4 // false
NaN >= 4 // false
NaN <= NaN // false
```

Те же операторы применяются для сравнения строк в лексикографическом порядке.

```
'Hello' < 'Goodbye' // false - H следует за G
'Hello' < 'Hi' // true - e предшествует i
```

При сравнении значений с помощью операторов `<`, `<=`, `>`, `>=` помните, что оба операнда должны быть числами или оба – строками. При необходимости преобразуйте операнды явно. В противном случае это сделает JavaScript, и иногда результаты могут оказаться неожиданными (см. следующий раздел).

Для сравнения на равенство служат следующие операторы:

`===` строгое равенство;
`!==` строгое неравенство.

С операторами строгого равенства все просто. Операнды разных типов никогда не равны строго. Значения `undefined` и `null` строго равны только самим себе. Числа, булевы значения и строки строго равны, если равны их значения.

```
'42' === 42 // false - разные типы
undefined === null // false
'42' === '4' + 2 // true - одна и та же строка '42'
```

Существуют также операторы «нестрогого равенства» `==` и `!=`, которые могут сравнивать значения разных типов. Вообще говоря, это не слишком полезно – детали см. в следующем разделе.

> **Предостережение.** Нельзя использовать выражение
>
> `x === NaN`
>
> для проверки того, что `x` равно `NaN`. Никакие два значения `NaN` не равны друг другу. Вместо этого пользуйтесь методом `Number.isNaN(x)`.

> **Примечание.** `Object.is(x, y)` почти то же самое, что `x === y`, с тем отличием, что `Object.is(+0, -0)` равно `false`, а `Object.is(NaN, NaN)` равно `true`.

Как в Java и Python, равенство объектов (включая массивы) означает, что оба операнда ссылаются на один и тот же объект. Ссылки на разные объекты никогда не равны, даже если содержимое объектов одинаково.

```
let harry = { name: 'Harry Smith', age: 42 }
let harry2 = harry
```

```
harry === harry2 // true - две ссылки на один и тот же объект
let harry3 = { name: 'Harry Smith', age: 42 }
harry === harry3 // false - разные объекты
```

2.6. Смешанное сравнение

Этот еще один раздел, отмеченный значком Безумного шляпника. В нем подробно описываются особенности JavaScript, способные посеять путаницу. Если вы будете следовать золотому правилу 3 – избегать сравнения разных типов, а в особенности держаться подальше от операторов «нестрогого равенства» (== и !=), то можете пропустить данный раздел.

Решили остаться? Сначала рассмотрим сравнение разных типов с помощью операторов <, <=, >, >=.

Если один операнд является числом, то другой преобразуется в число. Предположим, что другой операнд – строка. Преобразование дает числовое значение строки, если она содержит число, 0 – если строка пуста, и NaN – в противном случае. Кроме того, любое сравнение с NaN дает false – даже NaN <= NaN.

```
'42' < 5 // false - '42' преобразуется в число 42
'' < 5 // true - '' преобразуется в число 0
'Hello' <= 5 // false - 'Hello' преобразуется в NaN
5 <= 'Hello' // false - 'Hello' преобразуется в NaN
```

Теперь предположим, что второй операнд является массивом:

```
[4] < 5 // true - [4] преобразуется в число 4
[] < 5 // true - [] преобразуется в число 0
[3, 4] < 5 // false - [3, 4] преобразуется в NaN
```

Если оба операнда – не числа, то они преобразуются в строки. Такие сравнения редко дают осмысленный результат:

```
[1, 2, 3] < {} // true - [1, 2, 3] преобразуется в '1,2,3', {} в '[object Object]'
```

Теперь рассмотрим более пристально оператор нестрогого равенства. Работает он следующим образом.

- Если оба операнда одного типа, сравнить их строго.
- Значения undefined и null нестрого равны самим себе *и друг другу*, но не равны никаким другим значениям.
- Если один операнд – число, а другой – строка, преобразовать строку в число и сравнить строго.
- Если один операнд – булево значение, преобразовать оба операнда в числа и сравнить строго.
- Если один операнд – объект, а другой нет, преобразовать объект в примитивный тип (см. главу 8) и сравнить нестрого.

Например:

```
'' == 0 // true - '' преобразуется в 0
'0' == 0 // true - '0' преобразуется в 0
```

```
'0' == false // true - оба операнда преобразуются в 0
undefined == false // false - undefined равно только самому себе и null
```

Еще раз взглянем на строки '' и '0'. Обе «равны» 0. Но друг другу они не «равны»:

```
'' == '0' // false - преобразование не производится, т. к. оба операнда - строки
```

Как видим, правила нестрого сравнения не особенно полезны и легко могут стать причиной тонких ошибок. Не ходите в эту трясину, пользуйтесь операторами строгого сравнения (=== и !==).

> **Примечание.** Нестрогое сравнение x == null на самом деле истинно, если x равно undefined или null, а x != null – если x не равно ни тому, ни другому. Некоторые программисты, твердо решившие не пользоваться нестрогим равенством, делают исключение для этого случая.

2.7. Логические операторы

В JavaScript есть три оператора для комбинирования булевых значений:

- && и;
- || или;
- ! не.

Выражение x && y равно true, если x и y одновременно равны true, а x || y равно true, если хотя бы один из операндов x и y равен true. Выражение !x равно true, если x равно false.

Операторы && и || вычисляются лениво. Если левый операнд определяет результат (похож на false в случае &&, похож на true в случае ||), то правый операнд не вычисляется вовсе. Часто это оказывается полезно, например:

```
if (i < a.length && a[i] > 0) // a[i] > 0 не вычисляется, если i ≥ a.length
```

У операторов && и || есть еще одна любопытная особенность в случае, когда операнды не являются булевыми значениями. Они возвращают один из операндов в качестве значения выражения. Если левый операнд определяет результат, то он и становится значением выражения, а правый операнд не вычисляется. В противном случае значением выражения становится значение правого операнда.

Например:

```
0 && 'Harry' // 0
0 || 'Harry' // 'Harry'
```

Некоторые программисты, пытаясь обратить это поведение себе во благо, пишут примерно такой код:

```
let result = arg && arg.someMethod()
```

Идея в том, чтобы перед вызовом метода проверить, что arg не равен ни undefined, ни null. Если равен, то result также будет равен undefined или null. Но эта идиома трещит по швам, если arg равен нулю, пустой строке или false.

Еще одно применение – предоставить значение по умолчанию, когда метод возвращает undefined или null:

```
let result = arg.someMethod() || defaultValue
```

И снова идея оказывается несостоятельной, если метод может вернуть ноль, пустую строку или false.

Что нам действительно нужно, так это удобный способ использовать значение, если оно не равно ни undefined, ни null. По состоянию на начало 2020 года на третьей стадии рассмотрения находилось предложение двух операторов для этой цели. Это означает, что они, скорее всего, войдут в будущую версию JavaScript.

Выражение x ?? y равно x, если x не равно ни undefined, ни null, а в противном случае равно y. В выражении

```
let result = arg.someMethod() ?? defaultValue
```

значение по умолчанию defaultValue используется, только если метод возвращает undefined или null.

Выражение x?.*propertyName* возвращает значение свойства x с именем *propertyName*, если x не равно ни undefined, ни null, а в противном случае undefined. Рассмотрим выражение

```
let recipient = person?.name
```

Если person не равно ни undefined, ни null, то правая часть совпадает с person.name. Но если person равно undefined или null, то переменной recipient присваивается значение undefined. Если бы мы воспользовались оператором . вместо ?., то возникло бы исключение.

Операторы ?. можно сцеплять:

```
let recipientLength = person?.name?.length
```

Если person или person.name равно undefined или null, то recipientLength будет равно undefined.

> **Примечание.** В JavaScript имеются также поразрядные операторы & | ^ ~, которые сначала усекают свои операнды до 32-разрядного целого числа, а затем комбинируют их биты, в точности как аналогичные операторы в Java и C++. Существуют также операторы сдвига <<, >>, >>>, в которых левый операнд усекается до 32-разрядного целого, а правый – до 5-разрядного целого. Если вам нужно манипулировать отдельными битами 32-разрядных чисел, можете использовать эти операторы, в противном случае держитесь от них подальше.

> **Предостережение.** Некоторые программисты используют выражение x | 0, чтобы удалить дробную часть числа x. Результат получается неправильным, если $x \geq 2^{31}$. Лучше использовать для этой цели метод Math.floor(x).

2.8. Предложение switch

В JavaScript имеется предложение switch, работающее точно так же, как в C, C++, Java и C#, со всеми причудами. Если вы знакомы с предложением switch, пропустите этот раздел.

Предложение switch сравнивает выражение с несколькими возможными значениями, например:

```
let description = ''
switch (someExpression) {
  case 0:
    description = 'zero'
    break
  case false:
  case true:
    description = 'boolean'
    break
  case '':
    description = 'empty string' // см. предостережение ниже
  default:
    description = 'something else'
}
```

Выполнение начинается с метки case, которая строго равна значению выражения, и продолжается до следующего break или до конца предложения switch. Если ни с одной меткой нет совпадения, то выполнение начинается с метки default, если она присутствует.

Поскольку производится строгое сравнение на равенство, то в метках case не должно быть объектов.

> **Предостережение.** Если вы забудете поставить break в конце альтернативы, то выполнение продолжится для конца следующей альтернативы! В примере выше так происходит, когда value равно пустой строке. Сначала переменной description присваивается значение 'empty string', а потом 'something else'. Такое «проваливание» опасно и часто является причиной ошибок. Поэтому некоторые разработчики избегают предложения switch.

> **Совет.** Во многих случаях различие в производительности между switch и эквивалентным множеством предложений if пренебрежимо мало. Но если ветвей много, то виртуальная машина может построить «таблицу переходов» для эффективного перехода к нужной ветви case.

2.9. Циклы while и do

Это еще один раздел, который читатели, знакомые с C, C++, Java или C#, могут пропустить.

Цикл while выполняет предложение (которое может быть блоком), пока условие истинно. Общая форма такова:

```
while (условие) предложение
```

В следующем цикле определяется, за какое время удастся накопить определенную сумму денег для ухода на заслуженный отдых, если каждый год откладывать одну и ту же сумму денег, на которую начисляются проценты.

```
let years = 0
while (balance < goal) {
    balance += paymentAmount
    let interest = balance * interestRate / 100
    balance += interest
    years++
}
console.log(`${years} years.`)
```

Цикл `while` не исполняется ни разу, если условие в самом начале равно `false`. Если вы хотите, чтобы блок был выполнен хотя бы один раз, проверку нужно перенести в конец цикла, т. е. воспользоваться циклом `do-while` с таким синтаксисом:

```
do предложение while (условие)
```

Этот цикл выполняет предложение (которое может быть блоком), а затем проверяет условие. Если условие истинно, то предложение и проверка выполняются еще раз. Предположим, что мы только что обработали элемент `s[i]` и теперь ищем в строке следующий пробел.

```
do {
  i++
} while (i < s.length && s[i] != ' ')
```

После выхода из цикла либо `i` находится за пределами строки, либо `s[i]` равно пробелу.

Цикл `do` встречается гораздо реже, чем цикл `while`.

2.10. Циклы for

Цикл `for` – общая конструкция для обхода элементов. В следующих трех разделах обсуждаются варианты этого цикла, имеющиеся в JavaScript.

2.10.1. Классический цикл for

Классическая форма цикла `for` работает так же, как в C, C++, Java и C#. Имеется счетчик или еще какая-то переменная, которая обновляется после каждой итерации. В следующем цикле на консоль выводятся числа от 1 до 10:

```
for (let i = 1; i <= 10; i++)
  console.log(i)
```

В первой части предложения `for` производится инициализация счетчика. Во второй части записано условие, которое проверяется перед каждой

итерацией цикла. В третьей части указано, как обновляется счетчик после каждой итерации.

Как именно производится инициализация, проверка и обновление, зависит от решаемой задачи. Например, в следующем цикле элементы массива посещаются в обратном порядке:

```
for (let i = a.length - 1; i >= 0; i--)
  console.log(a[i])
```

> **Совет.** В первой части могут находиться произвольные объявления или выражения, а в остальных частях – произвольные выражения. Однако существует неписаное правило хорошего тона – инициализировать, проверять и обновлять нужно одну и ту же переменную.

> **Примечание.** В третьей части цикла for может присутствовать несколько выражений обновления, разделенных *оператором запятая*:
>
> ```
> for (let i = 0, j = a.length - 1; i < j; i++, j--) {
>   let temp = a[i]
>   a[i] = a[j]
>   a[j] = temp
> }
> ```
>
> В выражении `i++, j--` оператор запятая соединяет два выражения `i++` и `j--`, образуя новое выражение. Значением этого выражения является значение второго операнда. В данном случае значение не используется – нас интересуют только побочные эффекты инкремента и декремента.
>
> Оператор запятая мало кто любит, поскольку он может приводить к недоразумениям. Например, `Math.max((9, 3))` – максимум из одного значения (9, 3), т. е. 3.
>
> Запятая в объявлении `let i = 0, j = a.length - 1` – не оператор запятая, а синтаксическая часть предложения `let`. В этом предложении объявлены две переменные `i` и `j`.

2.10.2. Цикл for of

Цикл `for of` перебирает элементы *итерируемого объекта*, чаще всего массива или строки. (В главе 8 мы узнаем, как сделать итерируемыми другие объекты). Например:

```
let arr = [, 2, , 4]
arr[9] = 100
for (const element of arr)
  console.log(element) // печатается undefined, 2, undefined, 4,
                       // undefined (5 раз), 100
```

Цикл посещает все элементы массива с индексами от 0 до `arr.length - 1` в порядке возрастания. Для элементов с индексами 0, 2 и 4–8 печатается значение `undefined`.

Переменная `element` создается на каждой итерации цикла и инициализируется значением текущего элемента. Она объявлена как `const`, потому что в теле цикла не изменяется.

Цикл `for of` – приятное усовершенствование классического цикла `for` для случая, когда нужно обработать все элементы массива. Однако и у классиче-

ского цикла for остается много других применений. Например, когда нужно обойти не весь массив или требуется знать текущий индекс внутри цикла.

При обходе строки цикл for of посещает каждую *кодовую точку Юникода*. Именно такое поведение и желательно. Например:

```
let greeting = 'Hello 🌐'
for (const c of greeting)
  console.log(c) // печатается H e l l o, пробел и 🌐
```

Нам не нужно думать о том, что 🌐 занимает две кодовые единицы, хранящиеся в элементах greeting[6] и greeting[7].

2.10.3. Цикл for in

Цикл for of нельзя использовать для обхода значений свойств произвольного объекта, да это вряд ли и нужно – значения свойств обычно не имеют смысла без ключей. Вместо этого можно обойти ключи в цикле for in:

```
let obj = { name: 'Harry Smith', age: 42 }
for (const key in obj)
  console.log(`${key}: ${obj[key]}`)
```

В этом цикле будет напечатано age: 42 и name: Harry Smith в каком-то порядке.

В цикле for in перебираются ключи данного объекта. В главах 4 и 8 мы узнаем, что при переборе свойства «прототипа» включаются, а некоторые «неперечисляемые» свойства пропускаются. Порядок посещения ключей зависит от реализации, поэтому полагаться на него не следует.

> **Примечание.** В JavaScript цикл for of – то же самое, что «обобщенный» цикл for в Java, который также называют циклом «for each». У цикла for in в Java нет эквивалента.

Цикл for in можно использовать для перебора имен свойств массива.

```
let numbers = [1, 2, , 4]
numbers[99] = 100
for (const i in numbers)
  console.log(`${i}: ${numbers[i]}`)
```

Здесь i последовательно присваиваются значения '0', '1', '3' и '99'. Заметим, что, как и во всех объектах JavaScript, ключи свойств являются строками. Хотя в наиболее распространенных реализациях обход массивов производится в числовом порядке, лучше не полагаться на этот факт. Если порядок обхода имеет значение, то лучше воспользоваться циклом for of или классическим циклом for.

> **Предостережение.** Берегитесь выражений вида numbers[i + 1] в цикле for in. Например:
>
> ```
> if (numbers[i] === numbers[i + 1]) // Ошибка! i + 1 равно '01', '11' и т. д.
> ```

В этом условии сравниваются *не* соседние элементы. Поскольку `i` содержит строку, оператор + *конкатенирует строки*. Если `i` равно `'0'`, то `i + 1` равно `'01'`. Чтобы решить эту проблему, преобразуйте `i` в число:

```
if (numbers[i] === numbers[parseInt(i) + 1])
```

Или воспользуйтесь классическим циклом `for`.

Разумеется, если добавить в массив другие свойства, они тоже будут посещены:

```
numbers.lucky = true
for (const i in numbers) // i равно '0', '1', '3', '99', 'lucky'
  console.log(`${i}: ${numbers[i]}`)
```

В главе 4 мы узнаем, что кто-то третий может добавить перечисляемые свойства в `Array.prototype` или `Object.prototype`. Они тоже будут отражены в цикле `for in`. Поэтому этикет современного JavaScript очень косо смотрит на такую практику. Тем не менее некоторые программисты остерегают от использования цикла `for in`, поскольку опасаются унаследованных библиотек или коллег, которые могут скачать непонятно какой код из интернета.

Примечание. В следующей главе мы узнаем о другом способе обхода массива, с применением приемов функционального программирования. Например, можно вывести на консоль все элементы массива следующим образом:

```
arr.forEach((element, key) => { console.log(`${key}: ${element}`) })
```

Предоставленная функция вызывается для всех элементов и индексных ключей (в виде чисел 0 1 3 99, а не строк).

Предостережение. Когда цикл `for in` обходит строку, он посещает индексы каждой *кодовой единицы Юникода*. Может статься, что это не то, что вам нужно. Например:

```
let greeting = 'Hello 🌐'
for (const i of greeting)
  console.log(greeting[i])
    // Печатается H e l l o, пробел и два "битых" символа
```

Индексы 6 и 7 двух кодовых единиц символа Юникода посещаются по отдельности.

2.11. Break и continue

Иногда требуется выйти из цикла сразу после того, как цель достигнута. Допустим, что мы ищем позицию первого отрицательного элемента в массиве:

```
let i = 0
while (i < arr.length) {
  if (arr[i] < 0) ...
    ...
}
```

Встретив отрицательный элемент, мы хотим просто выйти из цикла, так чтобы переменная i содержала позицию элемента. Именно для этого и предназначено предложение break.

```
let i = 0
while (i < arr.length) {
  if (arr[i] < 0) break
  i++
}
// Сюда попадаем после break или в результате нормального завершения цикла
```

Без предложения break всегда можно обойтись. Можно добавить булеву переменную, которая будет управлять завершением цикла, часто ее называют done или found:

```
let i = 0
let found = false
while (!found && i < arr.length) {
  if (arr[i] < 0) {
    found = true
  } else {
    i++
  }
}
```

JavaScript, как и Java, включает предложение *break с меткой*, которое позволяет выйти из нескольких вложенных циклов. Пусть требуется найти положение первого отрицательного элемента в двумерном массиве. Когда элемент будет найден, нам понадобится выйти сразу из двух циклов. Добавим метку (т. е. идентификатор, за которым следует двоеточие) *перед* внешним циклом. Предложение break с меткой переходит на предложение, расположенное *после* помеченного цикла:

```
let i = 0
let j = 0
outer:
while (i < arr.length) {
  while (j < arr[i].length) {
    if (arr[i][j] < 0) break outer
      j++
    }
    i++
    j = 0
}
// Сюда попадаем после break outer или когда оба цикла нормально завершатся
```

Метка в предложении break с меткой должна находиться в той же строчке, что и ключевое слово break.

Предложения break с меткой встречаются нечасто.

Наконец, существует предложение continue, которое, как и break, нарушает нормальный поток управления. Оно передает управление в конец самого

внутреннего объемлющего цикла. В примере ниже вычисляется среднее значение положительных элементов массива:

```
let count = 0
let sum = 0
for (let i = 0; i < arr.length; i++) {
  if (arr[i] <= 0) continue
  count++
  sum += arr[i]
}
let avg = count === 0 ? 0 : sum / count
```

Если элемент не положительный, то предложение `continue` сразу передает управление на начало цикла, пропуская остаток *только текущей* итерации.

Если предложение `continue` встречается в цикле `for`, то оно передает управление на часть «обновления», как в этом примере.

Существует также форма предложения `continue` с меткой, которая передает управления в конец цикла с соответствующей меткой. Встречаются такие предложения редко.

Многие программисты считают, что предложения `break` и `continue` вносят хаос в программу. Без них легко можно обойтись, и в этой книге я их использовать не буду.

2.12. Перехват исключений

Некоторые методы возвращают индикатор ошибки, если вызываются с недопустимыми аргументами. Например, `parseFloat('')` возвращает значение `NaN`.

Но не всегда возврат индикатора ошибки – лучшее решение. Иногда нет очевидного способа отличить ошибочное значение от правильного. Хорошим примером служит метод `parseFloat`. Вызов `parseFloat('NaN')` возвращает `NaN`, а вызов `parseFloat('Infinity')` – значение `Infinity`. Если `parseFloat` вернул `NaN`, то нельзя сказать, что было тому причиной – разбор допустимой строки `'NaN'` или недопустимый аргумент.

В JavaScript метод может избрать другой путь, если завершить задачу без ошибок не удается. Вместо того чтобы возвращать значение, метод может *возбудить исключение*. В этом случае выполнение продолжается не в том коде, который вызвал метод, а в *ветви catch*. Если исключение нигде не перехвачено, то программа завершается.

Для перехвата исключения служит предложение `try`. В простейшем виде оно выглядит следующим образом:

```
try {
  код
  еще код
  еще код
} catch {
  обработчик
}
```

Если какой-нибудь код внутри блока try возбуждает исключение, то остаток кода в этом блоке пропускается, и управление передается коду обработчика внутри ветви catch.

Например, предположим, что мы получили JSON-строку и разбираем ее. Вызов функции JSON.parse возбуждает исключение, если аргумент – недопустимый текст в формате JSON. Обработаем эту ситуацию в ветви catch:

```
let input = ... // читать откуда-то входные данные
try {
  let data = JSON.parse(input)
  // Если выполнение дошло до этого места, значит, входные данные корректны
  // Обработать данные
  ...
} catch {
  // Обработать ошибку
  ...
}
```

Обработчик может записать информацию в журнал или предпринять еще какие-то действия, отражающие тот факт, что получена некорректная JSON-строка.

В главе 3 мы познакомимся с другими вариантами предложения try, которые позволяют более точно управлять процессом обработки исключения. Там же мы узнаем, как возбудить свое исключение.

Упражнения

1. На консоли браузера и в цикле REPL Node.js после выполнения предложений отображаются значения. Какие значения отображаются для следующих предложений:
   - предложение выражения;
   - объявление переменной;
   - предложение блока, содержащее хотя бы одно внутреннее предложение;
   - пустое предложение блока;
   - цикл while, do или for, тело которого выполняется хотя бы один раз;
   - цикл, тело которого не выполнено ни разу;
   - предложение if;
   - предложение try, которое завершается нормально;
   - предложение try, в котором выполнена ветвь catch?
2. Что не так со следующим предложением:

   ```
   if (x === 0) console.log('zero') else console.log('nonzero')
   ```

   Как исправить проблему?
3. Рассмотрим предложение

   ```
   let x = a
   ```

Какие лексемы в начале следующей строки предотвратят вставку точки с запятой? Какие из них могут встретиться в реальной программе?

4. Какие результаты дает сравнение значений `undefined`, `null`, `0` и `''` с помощью операторов `< <= ==`? Почему?
5. Всегда ли `a || b` совпадает с `a ? a : b` вне зависимости от типов `a` и `b`? Объясните свой ответ. Можно ли аналогичным образом выразить `a && b`?
6. Используйте все три вида цикла `for` для нахождения наибольшего значения в массиве чисел.
7. Рассмотрим следующий фрагмент кода:

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
for (i in arr) { if (i + 1 === 10) console.log(a[i]) }
```

Почему ничего не печатается?

8. Реализуйте предложение `switch`, которое конвертирует цифры от 0 до 9 в английские числительные от `'zero'` до `'nine'`. Как это можно легко сделать без `switch`? Что вы можете сказать об обратном преобразовании?
9. Пусть `n` – число от 0 до 7 и требуется присвоить элементам массива от `arr[k]` до `arr[k + n - 1]` значение 0. Воспользуйтесь предложением `switch` с проваливанием.
10. Перепишите цикл `do` из раздела 2.9 в виде цикла `while`.
11. Перепишите все циклы `for` из раздела 2.10 в виде циклов `while`.
12. Перепишите пример `break` с меткой из раздела 2.11 с использованием двух вложенных циклов `for`.
13. Перепишите пример `break` с меткой из раздела 2.11 без использования предложения `break`. Заведите булеву переменную, управляющую завершением вложенных циклов.
14. Перепишите пример `continue` из раздела 2.11 без использования предложения `continue`.
15. Рассмотрим задачу о нахождении первой позиции, начиная с которой массив `b` входит в массив `a` в виде подпоследовательности. Напишем два вложенных цикла:

```
let result = undefined
for (let i = 0; i < a.length - b.length; i++) {
 for (let j = 0; j < b.length; j++) {
   if (a[i + j] != b[j]) ...
 }
 ...
}
```

Завершите код, добавив предложения `break` и `continue` с метками. Затем перепишите его без использования `break` и `continue`.

Глава 3

Функции и функциональное программирование

В этой главе вы узнаете о том, как писать функции на JavaScript. JavaScript – «функциональный» язык программирования. Функции являются полноправными значениями, такими же, как числа и строки. Функции могут принимать и порождать другие функции. Овладение функциональным стилем программирования совершенно необходимо для работы с современным JavaScript.

Еще в этой главе рассматривается передача параметров и правила областей видимости, а также подробности возбуждения и перехвата исключений.

3.1. Объявление функций

В JavaScript для объявления функции нужно указать:

1) имя функции;
2) имена параметров;
3) тело функции, в котором вычисляется и возвращается ее результат.

Типы параметров и результата не задаются. Приведем пример.

```
function average(x, y) {
  return (x + y) / 2
}
```

Предложение `return` служит для возврата вычисленного функцией значения.

Чтобы вызвать эту функцию, нужно передать ей желаемые аргументы:

```
let result = average(6, 7) // результат равен 6.5
```

Что, если передать функции не число, а что-нибудь другое? Случится то, что и должно случиться, например:

```
result = average('6', '7') // результат равен 33.5
```

Когда передаются строки, оператор + в теле функции конкатенирует их. Получившаяся строка '67' преобразуется в число перед делением на 2.

Программисту на Java, C# или C++, привыкшему к проверке типов на этапе компиляции, это покажется небрежностью. Действительно, напутав с типами аргументов, мы узнаем об этом, только когда что-то странное произойдет на этапе выполнения. Зато, с другой стороны, можно писать функции, работающие с аргументами разных типов, это бывает удобно.

Предложение return возвращает управление немедленно, так что оставшаяся часть функции не выполняется. Для примера рассмотрим функцию indexOf, которая вычисляет индекс значения в массиве:

```
function indexOf(arr, value) {
  for (let i in arr) {
    if (arr[i] === value) return i
  }
  return -1
}
```

Как только искомое значение найдено, функция возвращает его индекс и завершается.

Функция может не возвращать никакого значения. Если в теле функции нет предложения return, или оно есть, но не сопровождается выражением, то функция возвращает значение undefined. Обычно так бывает, когда функция вызывается только ради побочного эффекта.

> **Совет.** Если иногда вы хотите, чтобы функция возвращала результат, а иногда – нет, то выражайте свои намерения явно:
>
> ```
> return undefined
> ```

> **Примечание.** Как было отмечено в главе 2, после предложения return в той же строке должна быть по крайней мере одна лексема, чтобы не произошло автоматической вставки точки с запятой. Например, если функция возвращает объект, то поставьте в той же строке хотя бы открывающую фигурную скобку:
>
> ```
> return {
>             average: (x + y) / 2,
>             max: Math.max(x, y),
>             . . .
> }
> ```

3.2. Функции высшего порядка

JavaScript – функциональный язык программирования. Функции являются значениями, которые можно сохранять в переменных, передавать в качестве аргументов или возвращать из функции в качестве значений.

Например, функцию average можно сохранить в переменной:

```
let f = average
```

А затем вызвать ее:

```
let result = f(6, 7)
```

При вычислении выражения `f(6, 7)` будет обнаружено, что `f` – функция, и эта функция будет вызвана с аргументами `6` и `7`.

Затем можно поместить в переменную `f` другую функцию:

```
f = Math.max
```

Если теперь вычислить `f(6, 7)`, то получим `7` – результат вызова `Math.max` с указанными аргументами.

Приведем пример передачи функции в качестве аргумента. Если `arr` – массив, то вызов метода

```
arr.map(someFunction)
```

применяет переданную функцию ко всем элементам и возвращает массив результатов (исходный массив при этом не изменяется). Например, в результате вызова

```
result = [0, 1, 2, 4].map(Math.sqrt)
```

`result` получает значение

```
[0, 1, 1.4142135623730951, 2]
```

Метод `map` иногда называют *функцией высшего порядка*: это функция, которая получает другую функцию в качестве параметра.

3.3. Функциональные литералы

Продолжим пример из предыдущего раздела. Пусть требуется умножить все элементы массива на 10. Конечно, можно написать функцию

```
function multiplyBy10(x) { return x * 10 }
```

И вызвать ее следующим образом:

```
result = [0, 1, 2, 4].map(multiplyBy10)
```

Но как-то обидно объявлять функцию только для того, чтобы использовать ее всего один раз. Лучше воспользоваться *функциональным литералом*. Приведем пример:

```
result = [0, 1, 2, 4].map(function (x) { return 10 * x })
```

Синтаксис прост. Используется та же синтаксическая конструкция `function`, что и прежде, только без имени функции. Функциональный литерал – это значение, которое обозначает функцию, выполняющую заданное действие. Это значение передается методу `map`. Сам по себе функциональный литерал

не имеет имени, как не имеет имени массивовый литерал `[0, 1, 2, 4]`. Если вы хотите придать функции имя, сделайте то же, что делаете всегда, когда хотите что-то поименовать, – сохраните ее в переменной.

```
const average = function (x, y) { return (x + y) / 2 }
```

> **Совет.** Анонимные функциональные литералы – это «нормальный» случай. Именованная функция – не что иное, как краткая запись объявления функционального литерала с последующим присвоением ему имени.

3.4. Стрелочные функции

В предыдущем разделе мы видели, как объявить функциональный литерал с помощью ключевого слова `function`. Существует еще одна, более краткая форма с использованием оператора `=>`, которая обычно называется «стрелочной»:

```
const average = (x, y) => (x + y) / 2
```

Переменные-параметры располагаются слева от стрелки, а возвращаемое значение – справа.

Если параметр всего один, то заключать его в скобки необязательно:

```
const multiplyBy10 = x => x * 10
```

Если у функции нет параметров, то следует оставить пустые скобки:

```
const dieToss = () => Math.trunc(Math.random() * 6) + 1
```

Отметим, что `dieToss` – функция, а не число. При каждом вызове `dieToss()` мы получаем случайное целое число от 1 до 6.

Если стрелочная функция более сложная, то ее тело следует поместить в блок. Для возврата значения из блока используется ключевое слово `return`:

```
const indexOf = (arr, value) => {
  for (let i in arr) {
    if (arr[i] === value) return i
  }
  return -1
}
```

> **Совет.** Лексема `=>` должна располагаться в одной строчке с параметрами:
>
> ```
> const average = (x, y) => // OK
>   (x + y) / 2
> const distance = (x, y) // Ошибка
>   => Math.abs(x - y)
> ```
>
> Если стрелочная функция занимает более одной строки, то лучше использовать фигурные скобки – так будет понятнее:
>
> ```
> const average = (x, y) => {
>   return (x + y) / 2
> }
> ```

Предостережение. Если стрелочная функция не делает ничего, а только возвращает объектный литерал, то необходимо заключить объект в круглые скобки:

```
const stats = (x, y) => ({
  average: (x + y) / 2,
  distance: Math.abs(x - y)
})
```

В противном случае фигурные скобки будут интерпретированы как блок.

Совет. В главе 4 мы увидим, что стрелочные функции обладают более регулярным поведением, чем функции, объявленные с помощью ключевого слова `function`. Многие программисты на JavaScript предпочитают использовать синтаксис со стрелкой, а не анонимные или вложенные функции. Одни используют стрелку для объявления всех вообще функций, тогда как другие употребляют слово `function` для объявления функций верхнего уровня. Это дело вкуса.

3.5. Функциональная обработка массива

Вместо того чтобы обходить массив в цикле `for of` или `for in`, можно воспользоваться методом `forEach`, передав ему функцию, которая получает на входе элемент и его индекс:

```
arr.forEach((element, index) => { console.log(`${index}: ${element}`) })
```

Эта функция вызывается для каждого элемента массива в порядке возрастания индексов.

Если вас интересуют только сами элементы, то можно передать функцию с одним параметром:

```
arr.forEach(element => { console.log(`${element}`) })
```

Метод `forEach` будет вызывать эту функцию, передавая элемент и индекс, но в данном случае индекс игнорируется.

Метод `forEach` не возвращает никакого результата. Но переданная ему функция должна иметь какой-то побочный эффект – напечатать значение или выполнить присваивание. Еще лучше, если есть возможность вообще избежать побочного эффекта и воспользоваться такими методами, как `map` или `filter`, которые преобразуют массив в желаемую форму.

В разделе 3.2 мы видели, что метод `map` преобразует массив, применяя к каждому элементу некоторую функцию. Приведем практически полезный пример. Пусть требуется построить HTML-список, содержащий элементы массива. Можно сначала окружить каждый элемент тегами `li`:

```
const enclose = (tag, contents) => `<${tag}>${contents}</${tag}>`
const listItems = items.map(i => enclose('li', i))
```

На самом деле безопаснее экранировать литеры `&` и `<`, встречающиеся в элементах. Предположим, что для этой цели у нас имеется функция `htmlEscape` (ее реализация приведена в сопроводительном коде к этой книге). Тогда

мы можем сначала преобразовать элементы массива, обезопасив их, а затем окружить тегами:

```
const listItems = items
  .map(htmlEscape)
  .map(i => enclose('li', i))
```

Теперь результатом является массив элементов, окруженных тегами `li`. Осталось конкатенировать все строки методом `Array.join` (см. главу 7) и заключить получившуюся строку в элемент `ul`:

```
const list = enclose('ul',
  items
  .map(htmlEscape)
  .map(i => enclose('li', i))
  .join(''))
```

У массива есть также полезный метод `filter`. Он получает на входе *предикат* – функцию, которая возвращает булево (или похожее на булево) значение. Результатом является массив всех элементов, удовлетворяющих этому предикату. Допустим, что в предыдущем примере мы не хотим включать в список пустые строки. Удалить их можно следующим образом:

```
const list = enclose('ul',
  items
  .filter(i => i.trim() !== '')
  .map(htmlEscape)
  .map(i => enclose('li', i))
  .join(''))
```

Эта конвейерная обработка – хороший пример высокоуровневого программирования в стиле «что, а не как сделать». Чего мы хотим? Отбросить пустые строки, экранировать HTML, заключить элементы массива в теги `li` и соединить их в одну строку. Как это делается? В конечном счете с помощью цепочки циклов и ветвлений, но это деталь реализации.

3.6. Замыкания

Функция `setTimeout` принимает два аргумента: функцию, которую нужно будет выполнить позже, когда истечет тайм-аут, и продолжительность тайм-аута в миллисекундах. Например, следующий вызов напечатает «Goodbye» через 10 секунд:

```
setTimeout(() => console.log('Goodbye'), 10000)
```

Сделаем это более гибко:

```
const sayLater = (text, when) => {
  let task = () => console.log(text)
  setTimeout(task, when)
}
```

Теперь можно вызвать эту функцию:

```
sayLater('Hello', 1000)
sayLater('Goodbye', 10000)
```

Обратите внимание на переменную `text` внутри стрелочной функции `() => console.log(text)`. Немного подумав, вы поймете, что происходит нечто совсем не очевидное. Код стрелочной функции выполняется спустя длительное время после возврата из функции `sayLater`. Но почему переменная `text` к этому моменту не уничтожилась? И как так получилось, что она сначала была равна `'Hello'`, а потом `'Goodbye'`?

Чтобы понять, в чем дело, нужно переосмыслить наше понимание функции. У функции есть три составляющие:

1) блок кода;
2) параметры;
3) свободные переменные, т. е. переменные, которые встречаются в коде, но не объявлены как параметры или локальные переменные.

Функция со свободными переменными называется *замыканием*.

В нашем примере `text` – свободная переменная стрелочной функции. В структуре данных, представляющей замыкание, хранится ссылка на переменную, существовавшую в момент создания функции. Мы говорим, что эта переменная *захвачена*. Поэтому ее значение доступно в момент, когда функция будет вызвана.

На самом деле стрелочная функция `() => console.log(text)` захватывает и еще одну переменную, а именно `console`.

Но каким образом `text` ухитряется получить два разных значения? Включим замедленное воспроизведение. При первом вызове `sayLater` создается замыкание, которое захватывает переменную, переданную в параметре `text` и имеющую значение `'Hello'`. По выходе из метода `sayLater` эта переменная не пропадает, поскольку все еще используется замыканием. При следующем вызове `sayLater` создается второе замыкание, которое захватывает другую переменную `text`, на этот раз имеющую значение `'Goodbye'`.

В JavaScript захваченная переменная является ссылкой на другую переменную, а не на текущее значение. Если изменить содержимое захваченной переменной, то это изменение будет видно в замыкании. Рассмотрим пример:

```
let text = 'Goodbye'
setTimeout(() => console.log(text), 10000)
text = 'Hello'
```

Через десять секунд печатается строка `'Hello'`, хотя в момент создания замыкания переменная `text` содержала строку `'Goodbye'`.

> **Примечание.** Лямбда-выражения и внутренние классы в Java тоже могут захватывать переменные из объемлющих областей видимости. Но в Java захваченная локальная переменная должна быть эффективно финальной, т. е. ее значение не может изменяться. Захват изменяемых переменных усложняет реализацию замы-

каний в JavaScript. Замыкание в JavaScript запоминает не только начальное значение, но и местоположение захваченной переменной. И захваченная переменная не уничтожается, пока существует замыкание, – даже если она была объявлена локальной в уже завершившемся методе.

Основная идея замыкания очень проста: свободная переменная внутри функции означает в точности то, что она же означает снаружи. Но последствия этой идеи огромны. Очень полезно захватывать переменные, в результате чего они остаются доступны сколь угодно долго. В следующем разделе мы приведем впечатляющую иллюстрацию, реализовав объекты и методы исключительно с помощью замыканий.

3.7. Крепкие объекты

Пусть требуется реализовать объекты, представляющие банковские счета. У каждого счета есть остаток. Денежные средства можно класть на счет и снимать со счета.

Мы хотим, чтобы состояние объекта было закрытым, т. е. чтобы его можно было модифицировать только с помощью предоставленных нами методов. Вот набросок фабричной функции:

```
const createAccount = () => {
  ...
  return {
    deposit: amount => { ... },
    withdraw: amount => { ... },
    getBalance: () => ...
  }
}
```

Теперь мы можем сконструировать столько счетов, сколько пожелаем:

```
const harrysAccount = createAccount()
const sallysAccount = createAccount()
sallysAccount.deposit(500)
```

Заметим, что объект счета содержит только методы, но не данные. Ведь если бы мы добавили остаток в объект счета, то его мог бы модифицировать кто угодно, поскольку в JavaScript нет закрытых свойств.

А где же хранить данные? Все просто – в виде локальных переменных фабричной функции:

```
const createAccount = () => {
  let balance = 0
  return {
    ...
  }
}
```

Мы захватываем локальные данные в методах:

```
const createAccount = () => {
  ...
  return {
    deposit: amount => {
      balance += amount
    },
    withdraw: amount => {
      if (balance >= amount)
        balance -= amount
      },
    getBalance: () => balance
  }
}
```

У каждого счета *своя собственная* захваченная переменная `balance` – та, которая была создана при вызове фабричной функции.

Мы можем передать фабричной функции параметры:

```
const createAccount = (initialBalance) => {
  let balance = initialBalance + 10 // бонус за открытие счета
  return {
    ...
  }
}
```

Можно даже захватить параметр вместо локальной переменной:

```
const createAccount = (balance) => {
    balance += 10 // бонус за открытие счета
    return {
      deposit: amount => {
      balance += amount
    },
    ...
  }
}
```

На первый взгляд, это довольно странный способ создания объектов. Но у таких объектов есть два важных преимущества. Состояние, включающее исключительно захваченные локальные переменные фабричной функции, автоматически инкапсулировано. И при этом мы избегаем параметра `this`, который, как станет ясно в главе 4, в JavaScript отнюдь не тривиален.

Эту технику иногда называют «паттерном замыкания» или «паттерном фабричного класса», но мне больше нравится термин, который Дуглас Крокфорд употребляет в книге «How JavaScript Works», – «крепкие объекты» (hard objects).

> **Примечание.** Чтобы еще сильнее укрепить объект, можно воспользоваться функцией `Object.freeze`, которая порождает объект, не допускающий ни модификации, ни удаления, ни добавления новых свойств.

```
const createAccount = (balance) => {
  return Object.freeze({
    deposit: amount => {
      balance += amount
    },
    ...
  })
}
```

3.8. Строгий режим

Мы уже видели, что в JavaScript хватает необычных возможностей, некоторые из которых, как оказалось, плохо приспособлены к разработке крупномасштабных проектов. *Строгий режим* ставит некоторые из этих возможностей вне закона, поэтому его всегда следует включать.

Чтобы включить строгий режим, поставьте первой отличной от комментария строкой в своем файле команду

```
'use strict'
```

Вместо одиночных кавычек допустимы двойные, точка с запятой тоже не возбраняется.

Чтобы включить строгий режим в REPL-цикле в Node.js, запустите его командой

```
node --use-strict
```

> **Примечание.** На консоли браузера каждую строку, которую требуется выполнить в строгом режиме, нужно предварять командой `'use strict';` или `'use strict'`, сопровождаемой нажатием **Shift**+**Enter**. Это не очень удобно.

Можно применить строгий режим к отдельным функциям:

```
function strictInASeaOfSloppy() {
  'use strict'
  ...
}
```

Нет никаких причин использовать в современном коде строгий режим для отдельных функций. Лучше включать его для всего файла.

Наконец, строгий режим включается внутри классов (см. главу 4) и в модулях ECMAScript (см. главу 10).

Для справки перечислим основные особенности строгого режима.

- Присваивание значения не объявленной ранее переменной считается ошибкой, при этом глобальная переменная не создается. Любую переменную следует объявлять с помощью одного из ключевых слов – `let`, `const` или `var`.
- Запрещается присваивать новое значение глобальному свойству, предназначенному только для чтения, например `NaN` или `undefined`. (Как это ни грустно, по-прежнему можно объявлять локальные переменные, затеняющие их.)
- Функции можно объявлять только на верхнем уровне скрипта или функции, но не во вложенном блоке.
- Оператор `delete` нельзя применять к «неквалифицированным идентификаторам». Например, `delete parseInt` является синтаксической ошибкой. Попытка удалить с помощью `delete` «неконфигурируемое» свойство (например, `delete 'Hello'.length`) приводит к ошибке во время выполнения.
- Запрещаются повторяющиеся параметры функции (`function average(x, x)`). Разумеется, это никогда и не нужно, но в нестрогом (прощающем небрежность) режиме это допустимо.
- Запрещаются восьмеричные литералы с префиксом `0`: `010` считается синтаксической ошибкой, а не восьмеричным числом 10 (десятичным 8). Если вам нужно восьмеричное число, пишите `0o10`.
- Запрещается предложение `with` (в этой книге оно не рассматривается).

> **Примечание.** В строгом режиме чтение необъявленной переменной приводит к исключению `ReferenceError`. Проверить, что переменная была объявлена (и инициализирована), с помощью конструкции
>
> `possiblyUndefinedVariable !== undefined`
>
> нельзя. Вместо этого используйте такую проверку:
>
> `typeof possiblyUndefinedVariable !== 'undefined'`

3.9. Проверка типов аргументов

В JavaScript не нужно задавать типы аргументов функции. Поэтому вызывающая сторона может передавать аргумент то одного, то другого типа, а функция будет обрабатывать его в соответствии с фактическим типом.

Так можно написать функцию `average`, которая готова принимать как числа, так и массив (хотя это несколько искусственный пример):

```
const average = (x, y) => {
  let sum = 0
  let n = 0
  if (Array.isArray(x)) {
    for (const value of x) { sum += value; n++ }
  } else {
    sum = x; n = 1
  }
```

```
  if (Array.isArray(y)) {
    for (const value of y) { sum += value }
  } else {
    sum += y; n++
  }
  return n === 0 ? 0 : sum / n
}
```

И вызывать ее следующим образом:

```
result = average(1, 2)
result = average([1, 2, 3], 4)
result = average(1, [2, 3, 4])
result = average([1, 2], [3, 4, 5])
```

В табл. 3.1 показано, как проверить принадлежность аргумента x тому или иному типу.

Таблица 3.1. Проверка типа

Тип	Проверка	Примечания
Строка	`typeof x === 'string' || x instanceof String`	x можно сконструировать как `new String(...)`
Регулярное выражение	`x instanceof RegExp`	
Число	`typeof x === 'number' || x instanceof Number`	x можно сконструировать как `new Number(...)`
Все, что может быть преобразовано в число	`typeof +x === 'number'`	Для получения числового значения нужно написать +x
Массив	`Array.isArray(x)`	
Функция	`typeof x === 'function'`	

Примечание. Некоторые программисты пишут функции, которые преобразуют значение любого аргумента в число, например:

```
const average = (x, y) => {
  return (+x + +y) / 2
}
```

Такую функцию можно вызвать следующим образом:

```
average('3', [4])
```

Полезна ли такая гибкость, безвредна ли она или является предвестником беды? Лично я не рекомендую так поступать.

3.10. Передача большего или меньшего числа аргументов

Предположим, что функция объявлена с определенным числом параметров, например:

```
const average = (x, y) => (x + y) / 2
```

На первый взгляд, при вызове этой функции необходимо передать два аргумента. Однако в JavaScript это не так. Функцию можно вызвать с большим числом аргументов, лишние будут молча проигнорированы:

```
let result = average(3, 4, 5) // 3.5 - последний аргумент игнорируется
```

Наоборот, если передать меньше аргументов, то недостающим будет присвоено значение `undefined`. Например, `average(3)`равно `(3 + undefined) / 2` или `NaN`. Если вы хотите поддержать такой вызов и получить осмысленный результат, то можете написать:

```
const average = (x, y) => y === undefined ? x : (x + y) / 2
```

3.11. Аргументы по умолчанию

В предыдущем разделе мы видели, как реализовать функцию, которую можно вызывать с меньшим числом фактических аргументов, чем имеется формальных параметров. Вместо того чтобы вручную проверять, равен ли аргумент `undefined`, можно предоставить параметры по умолчанию в объявлении функции. После параметра добавьте знак `=` и выражение, подразумеваемое по умолчанию, – оно будет использоваться, если аргумент не передан.

Вот еще один способ заставить функцию `average` работать с одним аргументом:

```
const average = (x, y = x) => (x + y) / 2
```

Если вызвать `average(3)`, то `y` получит значение `x`, т. е. `3`, и будет возвращен правильный результат.

Значений по умолчанию может быть несколько:

```
const average = (x = 0, y = x) => (x + y) / 2
```

Теперь `average()` возвращает 0.

Можно даже предоставить значение по умолчанию для первого параметра и не предоставлять для других:

```
const average = (x = 0, y) => y === undefined ? x : (x + y) / 2
```

Если аргумент (в т. ч. равный `undefined`) не передан, то параметру присваивается значение по умолчанию, а если его нет, то `undefined`:

```
average(3) // average(3, undefined)
average() // average(0, undefined)
average(undefined, 3) // average(0, 3)
```

3.12. Прочие параметры и оператор расширения

Как мы видели, функцию в JavaScript можно вызывать с любым числом аргументов. Чтобы обработать их все, объявите в качестве последнего параметра функции «прочие», предварив его лексемой …:

```
const average = (first = 0, ...following) => {
  let sum = first
  for (const value of following) { sum += value }
  return sum / (1 + following.length)
}
```

При вызове этой функции параметр `following` является массивом, который содержит все фактические аргументы, которые не использовались для инициализации предыдущих формальных параметров. Например, рассмотрим вызов

```
average(1, 7, 2, 9)
```

Тогда `first` будет равно `1`, а `following` – массиву `[7, 2, 9]`.

Многие функции и методы принимают переменное число аргументов. Например, функция `Math.max` возвращает наибольший из переданных аргументов, сколько бы их ни было:

```
let result = Math.max(3, 1, 4, 1, 5, 9, 2, 6) // result равно 9
```

А что, если значением уже является массив?

```
let numbers = [1, 7, 2, 9]
result = Math.max(numbers) // возвращает NaN
```

Это не работает. Метод `Math.max` получает массив с одним элементом – массивом `[1, 7, 2, 9]`.

Вместо этого воспользуйтесь оператором «расширения» – поставьте лексему … перед массивовым *аргументом*:

```
result = Math.max(...numbers) // возвращает 9
```

Оператор расширения линеаризует элементы, как если бы они передавались по отдельности.

> **Примечание.** Хотя оператор расширения и объявление прочих параметров выглядят одинаково, по существу они в точности противоположны.
> Прежде всего отметим, что оператор расширения применяется к аргументу, тогда как синтаксис «прочих» относится к объявлению переменной.
>
> ```
> Math.max(...numbers) // оператор расширения – аргумент при вызове функции
> const max = (...values) => { /* тело */}
>   // объявление прочих параметров
> ```
>
> Оператор расширения преобразует массив (вообще-то, любой итерируемый объект) в последовательность значений. Объявление «прочих» приводит к тому, что последовательность значений помещается в массив.

Заметим, что оператор расширения можно использовать, даже если у вызываемой функции нет «прочих» параметров. Например, рассмотрим функцию `average` с двумя параметрами из предыдущего раздела. Если вызвать

```
result = average(...numbers)
```

то все элементы массива `numbers` передаются функции в виде аргументов. Функция использует первые два аргумента, а остальные игнорирует.

> **Примечание.** Оператор расширения можно использовать также в инициализаторе массива:
>
> `let moreNumbers = [1, 2, 3, ...numbers] // оператор расширения`
>
> Не путайте это с объявлением прочих параметров, используемым при деструктуризации. Объявление «прочих» применяется к переменной:
>
> `let [first, ...following] = numbers // объявление прочих параметров`

> **Совет.** Поскольку строки допускают итерирование, оператор расширения можно применить и к строке:
>
> `let greeting = 'Hello 🌐'`
> `let characters = [...greeting]`
>
> Массив `characters` содержит строки `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, `' '` и `'🌐'`.
> В объявлении функции можно одновременно использовать синтаксис аргументов по умолчанию и прочих параметров.
>
> `function average(first = 0, ...following) { . . . }`

3.13. Имитация именованных аргументов с помощью деструктуризации

В JavaScript нет конструкции «именованных аргументов», позволяющей при вызове функции указывать имена параметров. Но ее можно легко имитировать, передав объектный литерал:

```
const result = mkString(values, { leftDelimiter: '(', rightDelimiter: ')' })
```

Для вызывающей стороны тут нет ничего сложного. А теперь обратимся к реализации. Мы можем поискать имя среди свойств объекта, а если его там нет, подставить значение по умолчанию.

```
const mkString = (array, config) => {
  let separator = config.separator === undefined ? ',' : config.separator
  ...
}
```

Однако это утомительно. Проще использовать деструктурированные параметры с умолчаниями (см. описание синтаксиса деструктуризации в главе 1).

```
const mkString = (array, {
    separator = ',',
    leftDelimiter = '[',
```

```
    rightDelimiter = ']'
  }) => {
  ...
}
```

Синтаксис деструктуризации { **separator** = ',', **leftDelimiter** = '[', **rightDelimiter** = ']' } позволяет объявить три параметра: `separator`, `leftDelimiter` и `rightDelimiter` – и инициализировать их значениями одноименных свойств. Если некоторое свойство отсутствует или равно `undefined`, то вместо него подставляется значение по умолчанию.

Разумно задавать значение по умолчанию {} для объекта конфигурации:

```
const mkString = (array, {
  separator = ',',
  leftDelimiter = '[',
  rightDelimiter = ']'
} = {}) => {
  ...
}
```

Теперь функцию можно вызывать вообще без объекта конфигурации:

```
const result = mkString(values) // второй аргумент по умолчанию равен {}
```

3.14. Поднятие

В этом разделе, помеченном значком Безумного шляпника, мы рассмотрим еще один сложный вопрос, который легко можно обойти, придерживаясь трех простых правил:

- не использовать `var`;
- использовать строгий режим;
- объявлять переменные и функции до использования.

Если вы хотите знать, что бывает, когда этим правилам не следуют, читайте дальше.

В JavaScript имеется необычный механизм определения *области видимости* переменной – той части программы, в которой к переменной можно обращаться. Рассмотрим локальную переменную, объявленную внутри функции. В таких языках программирования, как Java, C# или C++, область видимости простирается от точки объявления переменной до конца объемлющего блока. В JavaScript локальная переменная, объявленная с помощью `let`, на первый взгляд, обладает таким же поведением:

```
function doStuff() { // начало блока
  ... // при попытке доступа к someVariable возбуждается ReferenceError
  let someVariable // область видимости начинается здесь
  ... // можно обратиться к someVariable, значение равно undefined
  someVariable = 42
  ... // можно обратиться к someVariable, значение равно 42
} // конец блока, здесь область видимости заканчивается
```

Однако все не так просто. К локальным переменным *можно* обращаться в функциях, объявления которых предшествуют объявлению переменной:

```
function doStuff() {
  function localWork() {
    console.log(someVariable) // обратиться к переменной можно
    ...
  }
  let someVariable = 42
  localWork() // печатается 42
}
```

В JavaScript любое объявление *поднимается* наверх его области видимости. То есть еще до объявления известно, что переменная или функция существует, и для ее значения резервируется место.

Внутри вложенной функции можно ссылаться на поднятые переменные или функции. Рассмотрим функцию `localWork` из предыдущего примера. Эта функция знает о местоположении переменной `someVariable`, потому что та поднята в начало тела функции `doStuff`, хотя переменная объявлена после `localWork`.

Конечно, может случиться, что обращение к переменной предшествует выполнению предложения, в котором она объявлена. Если переменная объявлена с помощью ключевого слова `let` или `const`, то такое обращение приведет к исключению `ReferenceError`. Переменная находится во «временно мертвой зоне», пока не будет произведено объявление.

Но если переменная объявлена с помощью архаичного ключевого слова `var`, то она просто равна `undefined`, до тех пор пока не будет инициализирована.

> **Совет.** Не пользуйтесь ключевым словом `var`. Оно объявляет переменные, областью видимости которых является вся функция, а не только объемлющий блок. Это слишком широко:
>
> ```
> function someFunction(arr) {
>   // i, element уже находятся в области видимости, но равны undefined
>   for (var i = 0; i < arr.length; i++) {
>     var element = arr[i]
>     ...
>   }
>   // i, element все еще в области видимости
> }
> ```
>
> Кроме того, `var` плохо сочетается с замыканиями, см. упражнение 10.

Поскольку функции поднимаются, мы можем вызывать функцию еще до того, как она объявлена. В частности, можно объявить взаимно рекурсивные функции:

```
function isEven(n) { return n === 0 ? true : isOdd(n -1) }
function isOdd(n) { return n === 0 ? false : isEven(n -1) }
```

Примечание. В строгом режиме именованные функции могут быть объявлены только на верхнем уровне скрипта или функции, но не внутри вложенного блока. В нестрогом режиме вложенные именованные функции поднимаются наверх объемлющей функции. В упражнении 12 показано, почему это плохая идея.

Если вы включите строгий режим и будете избегать объявлений `var`, то поднятие вряд ли приведет к ошибкам в программе. Однако рекомендуется структурировать программу так, чтобы переменные и функции объявлялись до использования.

Примечание. Когда-то давным-давно программисты на JavaScript использовали «немедленно вызываемые функции», чтобы ограничить область видимости `var`-объявлений переменных и функций:

```
(function () {
  var someVariable = 42
  function someFunction(...) { ... }
    ...
})() // функция вызывается здесь – обратите внимание на ()
// someVariable, someFunction больше не находятся в области видимости
```

После вызова анонимная функция больше никогда не используется. Ее единственное назначение – инкапсулировать объявления.

Этот механизм больше не нужен. Просто напишите:

```
{
  let someVariable = 42
  const someFunction = (...) => { ... }
  ...
}
```

Эти объявления ограничены блоком.

3.15. Возбуждение исключений

Если функция не может вычислить результат, она вправе возбудить исключение. В зависимости от характера ошибки это может оказаться лучшей стратегией, чем возврат кода ошибки, например `NaN` или `undefined`.

Для возбуждения исключения служит предложение `throw`:

```
throw value
```

Значение исключения может иметь любой тип, но традиционно в этом качестве используется объект ошибки. Функция `Error` порождает такой объект, содержащий строку с описанием причины ошибки.

```
let reason = `Элемент ${elem} не найден`
throw Error(reason)
```

После выполнения предложения `throw` функция немедленно завершается. Не возвращается никакого значения, даже `undefined`. Выполнение продолжается не с места вызова функции, а в ближайшей ветви `catch` или `finally`, как описано в следующих разделах.

Совет. Обработка исключений – удобный механизм для непредвиденных ситуаций, которые вызывающая сторона не умеет обрабатывать. Но он не годится для ситуаций, когда ошибка ожидаема. Рассмотрим разбор данных, введенных пользователем. Весьма вероятно, что некоторые пользователи будут вводить неправильные данные. В JavaScript в таких случаях легко вернуть индикатор ошибки, например `undefined`, `null` или `NaN` (при условии, конечно, что он не совпадает с допустимыми данными). Или можно вернуть объект, описывающий успех либо неудачу. Например, в главе 9 мы встретим метод, который возвращает объекты вида { `status: 'fulfilled', value:` *результат* } или { `status: 'rejected', reason:` *исключение* }.

3.16. Перехват исключений

Для перехвата исключений предназначено предложение `try`. В главе 2 мы видели, как перехватить исключение, если его значение нас не интересует. Если же вы хотите проанализировать значение исключения, то добавьте переменную в ветвь `catch`:

```
try {
  // Сделать что-то
  ...
} catch (e) {
  // Обработать исключения
  ...
}
```

Переменная в ветви `catch` (здесь `e`) содержит значение исключения. В предыдущем разделе мы видели, что, по соглашению, значением исключения является объект ошибки. У такого объекта есть два свойства: `name` и `message`. Например, если вызвать

```
JSON.parse('{ age: 42 }')
```

то будет возбуждено исключение с именем `'SyntaxError'` и сообщением `'Unexpected token a in JSON at position 2'`. (В этом примере строка является недопустимой в JSON, потому что ключ `age` не заключен в двойные кавычки.)

Объект, созданный функцией `Error`, называется `'Error'`. Виртуальная машина JavaScript возбуждает исключения с именами `'SyntaxError'`, `'TypeError'`, `'RangeError'`, `'ReferenceError'`, `'URIError'` или `'InternalError'`.

В обработчике мы можем сохранить эту информацию в подходящем месте. Однако в JavaScript обычно не имеет смысла детально анализировать объект ошибки, как в языках типа Java или C++.

Если вывести объект ошибки на консоль, то исполняющая среда JavaScript, как правило, отображает *трассу стека* – вызовы функций и методов между точками возбуждения и перехвата исключения. К сожалению, не существует стандартного способа доступа к трассе стека с целью ее протоколирования в другом месте.

> **Примечание.** В Java и C++ можно перехватывать исключения по типу. Это позволяет обработать ошибки некоторых типов на нижнем уровне, а остальные – на верхнем. Реализовать такую стратегию в JavaScript нелегко. Ветвь `catch` перехватывает *все* исключения, а объект исключения несет недостаточно информации. В JavaScript обработчики исключений обычно производят общее восстановление или очистку, не пытаясь анализировать причину ошибки.

При входе в ветвь `catch` исключение считается обработанным. Обработка продолжается с этой точки, т. е. выполняются предложения в ветви `catch`. Из ветви `catch` можно выйти с помощью предложения `return` или `break` либо дойдя до ее конца. В последнем случае управление передается следующему за `catch` предложению.

Если исключения протоколируются на одном уровне программы, а обработка исключения производится на более высоком уровне, то необходимо *повторно возбудить* исключение после протоколирования:

```
try {
  // Что-то сделать
  ...
} catch (e) {
  console.log(e)
  throw e // Повторно возбудить для обработки ошибки
}
```

3.17. Ветвь finally

В предложении `try` может быть необязательная ветвь `finally`. Код в ветви `finally` выполняется вне зависимости от того, произошло исключение или нет.

Начнем с простейшего случая: предложение `try` с ветвью `finally`, но без ветви `catch`:

```
try {
  // Захватить ресурсы
  ...
  // Что-то сделать
  ...
} finally {
  // Освободить ресурсы
  ...
}
```

Ветвь `finally` выполняется в следующих случаях:

- если все предложения в ветви `try` завершились без возбуждения исключения;
- если в ветви `try` было выполнено предложение `return` или `break`;
- если в каком-нибудь предложении в ветви `try` возникло исключение.

Возможно также предложение try с обеими ветвями catch и finally:

```
try {
  ...
} catch (e) {
  ...
} finally {
  ...
}
```

Теперь появился дополнительный путь. Если в ветви try возникло исключение, то выполняется ветвь catch. Вне зависимости от способа выхода из catch (нормально или с помощью return/break/throw) далее выполняется ветвь finally.

Назначение ветви finally – организовать единственное место для освобождения ресурсов (например, описателей файлов или подключений к базе данных), захваченных в ветви try, – не важно, произошло исключение или нет.

> **Предостережение.** Допустимо включать предложения return/break/throw в ветвь finally, хотя их эффект интуитивно не очевиден. Эти предложения отменяют действие предложений в ветвях try и catch. Например:
>
> ```
> try {
>   // Что-то сделать
>   ...
>   return true
> } finally {
>   ...
>   return false
> }
> ```
>
> Если блок try завершился успешно и было выполнено предложение return true, то дальше выполняется ветвь finally. Предложение return false в ней отменяет прежнее предложение return.

Упражнения

1. Что делает функция indexOf из раздела 3.1, когда вместо массива ей передается объект?
2. Перепишите функцию indexOf из раздела 3.1, так чтобы возврат управления производился только в конце.
3. Напишите функцию values(f, low, high), которая возвращает массив значений функции [f(low), f(low + 1), ..., f(high)].
4. Метод sort массива может принимать в качестве аргумента функцию сравнения с двумя параметрами, назовем их x и y. Функция возвращает отрицательное целое число, если x должен предшествовать y, ноль, если x и y одинаковы, и положительное число, если x должен следовать за y. Пользуясь стрелочными функциями, напишите вызовы метода sort, которые сортируют:
   - массив положительных целых чисел в порядке убывания;

- массив людей в порядке возрастания возраста;
- массив строк в порядке возрастания длины.

5. Пользуясь «крепкими объектами» из раздела 3.7, реализуйте метод `constructCounter`, порождающий объекты счетчика, метод `count` которых увеличивает счетчик и возвращает новое значение. Начальное значение и необязательная величина инкремента передаются в качестве параметров (по умолчанию инкремент равен 1).

```
const myFirstCounter = constructCounter(0, 2)
console.log(myFirstCounter.count()) // 0
console.log(myFirstCounter.count()) // 2
```

6. Программист думает, что «именованные параметры почти реализованы в JavaScript, но порядок все-таки имеет приоритет», и предлагает следующее «подтверждение» на консоли браузера:

```
function f(a=1, b=2){ console.log(`a=${a}, b=${b}`) }
f() // a=1, b=2
f(a=5) // a=5, b=2
f(a=7, b=10) // a=7, b=10
f(b=10, a=7) // нужно передавать по порядку: a=10, b=7
```

   Что тут происходит на самом деле? (Указание: это не имеет никакого отношения к именованным параметрам. Попробуйте выполнить код в строгом режиме.)
7. Напишите функцию `average`, которая вычисляет среднее значение произвольной последовательности чисел, воспользовавшись синтаксисом прочих параметров.
8. Что будет, если передать строковый аргумент в качестве прочих параметров `...str`? Приведите пример, показывающий, как с пользой применить это наблюдение.
9. Доведите до конца функцию `mkString` из раздела 3.13.
10. Архаичное ключевое слово `var` плохо сочетается с замыканиями. Рассмотрим пример:

```
for (var i = 0; i < 10; i++) {
  setTimeout(() => console.log(i), 1000 * i)
}
```

    Что напечатает этот код? Почему? (Указание: какова область видимости переменной `i`?) Какое простое изменение нужно внести, чтобы печатались числа 0, 1, 2, ..., 9?
11. Рассмотрим следующее объявление функции факториал:

```
const fac = n => n > 1 ? n * fac(n - 1) : 1
```

    Объясните, почему оно работает только благодаря поднятию переменной.
12. В нестрогом режиме функции можно объявлять внутри вложенного блока, и они поднимаются в начало объемлющей функции или блока. Попробуйте несколько раз выполнить следующий код:

```
if (Math.random() < 0.5) {
    say('Hello')
    function say(greeting) { console.log(`${greeting}!`) }
}
say('Goodbye')
```

Как результат зависит от значения, возвращаемого методом `Math.random`? Какова область видимости `say`? Когда она инициализируется? Что произойдет, если включить строгий режим?

13. Реализуйте функцию `average`, которая возбуждает исключение, если какой-нибудь из ее аргументов не является числом.
14. Некоторые программисты путаются в предложениях, содержащих все три ветви `try/catch/finally`, потому что путей управления слишком много. Покажите, как можно переписать такое предложение, используя только `try/catch` или только `try/finally`.

Глава 4

Объектно-ориентированное программирование

Как вы знаете, в JavaScript есть объекты, но они не похожи на объекты, которые мы привыкли видеть в объектно-ориентированных языках программирования типа Java или C++. Все свойства объекта в JavaScript открыты, а сами объекты не принадлежат никакому классу, кроме `Object`. Не понятно, как реализовать методы, классы и наследование.

Но все это *можно* устроить в JavaScript, и в этой главе показано, как именно. В современных версиях JavaScript имеется синтаксис для объявления классов, внешне очень похожих на классы в Java, хотя внутренний механизм совершенно иной. Необходимо понимать, что творится под капотом. Поэтому я сначала покажу, как объявлять методы и функции-конструкторы вручную, а потом мы посмотрим, как эти конструкции отображаются на синтаксис классов.

4.1. Методы

JavaScript, в отличие от большинства объектно-ориентированных языков, позволяет работать с объектами, не объявляя предварительно классов. Мы уже видели, как создаются объекты:

```
let harry = { name: 'Harry Smith', salary: 90000 }
```

Согласно классическому определению, у объекта имеется идентичность, состояние и поведение. Только что показанный объект, безусловно, обладает идентичностью – он отличается от любого другого объекта. Состояние объекта определяется его свойствами. Добавим поведение в виде «метода», т. е. свойства, значением которого является функция:

```
harry = {
  name: 'Harry Smith',
  salary: 90000,
```

```
  raiseSalary: function(percent) {
    this.salary *= 1 + percent / 100
  }
}
```

Теперь можно повысить зарплату работнику, применив знакомую нотацию с точкой:

```
harry.raiseSalary(10)
```

Заметим, что `raiseSalary` – функция, объявленная в объекте `harry`. Выглядит она, как любая другая функция, но с одним отличием: в ее теле мы обращаемся к `this.salary`. При вызове функции `this` ссылается на объект, указанный слева от точки.

Для объявления методов существует краткий синтаксис. Можно опустить двоеточие и ключевое слово `function`:

```
harry = {
  name: 'Harry Smith',
  salary: 90000,
  raiseSalary(percent) {
    this.salary *= 1 + percent / 100
  }
}
```

Выглядит похоже на объявление методов в Java или C++, но это всего лишь «синтаксический сахар» для свойств со значениями-функциями.

> **Предостережение.** Ссылка `this` работает только в функциях, объявленных с помощью слова `function` или краткого синтаксиса, в котором `function` опущено, но не в стрелочных функциях. Дополнительные сведения см. в разделе 4.12 «Ссылка this».

4.2. Прототипы

Предположим, что имеется много объектов работников, похожих на показанный в предыдущем разделе. И мы хотим включить свойство `raiseSalary` в каждый из них. Для автоматизации этой задачи можно написать фабричную функцию:

```
function createEmployee(name, salary) {
  return {
    name: name,
    salary: salary,
    raiseSalary: function(percent) {
      this.salary *= 1 + percent / 100
    }
  }
}
```

Но все равно у каждого объекта работника имеется собственное свойство `raiseSalary`, пусть даже его значениями являются одинаковые функции (см. рис. 4.1). Было бы лучше, если бы все работники разделяли одну общую функцию.

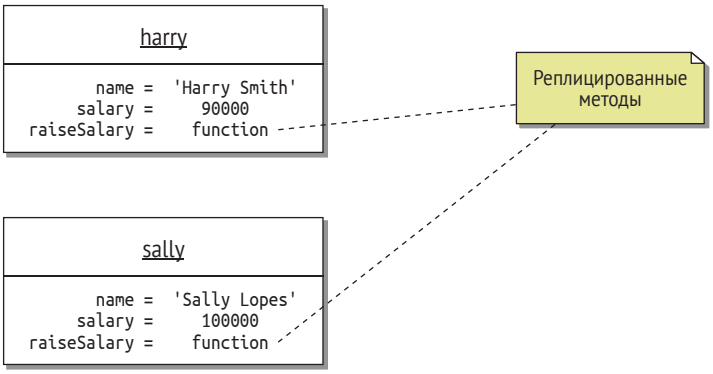

Рис. 4.1 ❖ Объекты с реплицированными методами

Именно для этого предназначены *прототипы*. В прототипе собраны свойства, общие для нескольких объектов. Вот как выглядит прототип, содержащий разделяемые методы:

```
const employeePrototype = {
  raiseSalary: function(percent) {
    this.salary *= 1 + percent / 100
  }
}
```

При создании объекта работника мы задаем его прототип. Прототип – это «внутренний слот» объекта. Этот технический термин употребляется в спецификации языка ECMAScript для обозначения атрибута объекта, над которым производятся внутренние манипуляции, но который не раскрывается программистам в виде свойства. Читать и изменять внутренний слот [[Property]] (так он называется в спецификации) позволяют методы `Object.getPrototypeOf` и `Object.setPrototypeOf`. Следующая функция создает объект работника и устанавливает его прототип:

```
function createEmployee(name, salary) {
  const result = { name, salary }
  Object.setPrototypeOf(result, employeePrototype)
  return result
}
```

На рис. 4.2 показан результат создания нескольких объектов работников с общим прототипом. Слот прототипа обозначен [[Prototype]], как в спецификации ECMAScript.

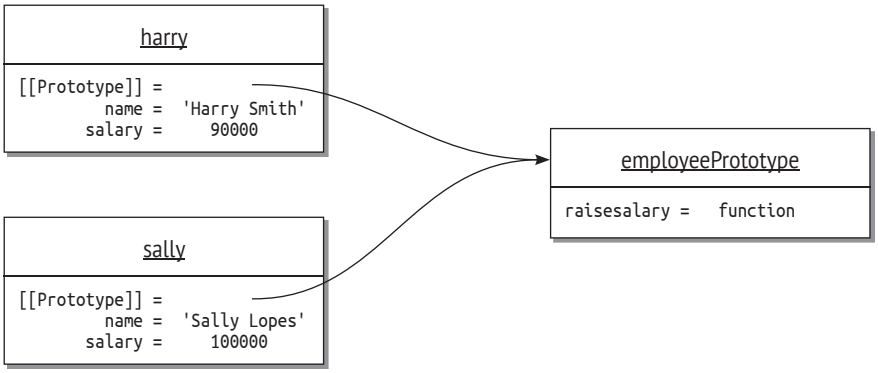

Рис. 4.2 ❖ Объекты с общим прототипом

Предостережение. Во многих реализациях JavaScript к прототипу объекта можно обратиться с помощью свойства `obj.__proto__`. Это не стандартный способ, следует использовать методы `Object.getPrototypeOf` и `Object.setPrototypeOf`.

Теперь рассмотрим вызов метода

```
harry.raiseSalary(5)
```

В самом объекте `harry` свойства `harry.raiseSalary` нет, поэтому его поиск продолжается в прототипе. Так как в `harry.[[Prototype]]` есть свойство `raiseSalary`, его значение и становится значением `harry.raiseSalary`.

Ниже в этой главе мы увидим, что прототипы можно сцеплять. Если в некотором прототипе нет свойства, то производится поиск в его прототипе, и так до конца цепочки прототипов.

Механизм поиска в прототипах весьма общий. Здесь мы использовали его для поиска метода, но работает он для любого свойства. Если свойство не найдено в самом объекте, то производится поиск в прототипе, и первое совпадение является значением свойства.

Поиск в прототипе – простая, но очень важная в JavaScript идея. Прототипы используются для реализации классов и наследования, а также для модификации поведения объектов уже после создания.

> **Примечание.** Поиск в цепочке прототипов используется только для *чтения* значений свойств. При записи в свойство всегда обновляется значение в самом объекте.

Пусть, например, мы изменили метод `harry.raiseSalary`:

```
harry.raiseSalary = function(rate) { this.salary = Number.MAX_VALUE }
```

При этом новое свойство добавляется в сам объект `harry`. Прототип не модифицируется, поэтому у всех работников сохраняется оригинальное свойство `raiseSalary`.

4.3. Конструкторы

В предыдущем разделе мы видели, как написать фабричную функцию, которая создает новые экземпляры объектов с общим прототипом. Для вызова таких функций существует специальный синтаксис с оператором `new`.

По соглашению, функции, конструирующие объекты, называют так, как назвали бы класс в языке, где классы существуют. В нашем примере назовем функцию `Employee`:

```
function Employee(name, salary) {
    this.name = name
    this.salary = salary
}
```

При вызове

```
new Employee('Harry Smith', 90000)
```

оператор `new` создает новый пустой объект, а затем вызывает функцию-конструктор. Параметр `this` указывает на вновь созданный объект. В теле функции `Employee` устанавливаются свойства объекта с помощью параметра `this`. Этот объект становится значением выражения `new`.

> **Предостережение.** Не возвращайте результат из функции-конструктора. В противном случае значением выражения `new` станет это возвращенное значение, а не вновь созданный объект.

Помимо вызова конструктора, выражение `new` выполняет еще одно важное действие: заполняет внутренний слот [[Prototype]]. В этот слот записывается специальный объект, прикрепленный к функции-конструктору. Напомним, что функция является объектом, поэтому может иметь свойства. У каждой *функции* в JavaScript имеется свойство `prototype`, значением которого является объект.

Этот объект предоставляет готовое место для добавления методов, например:

```
Employee.prototype.raiseSalary = function(percent) {
  this.salary *= 1 + percent / 100
}
```

Как видим, происходит много интересного. Еще раз взглянем на вызов:

```
const harry = new Employee('Harry Smith', 90000)
```

Распишем детально все шаги.

1. Оператор `new` создает новый объект.
2. Во внутренний слот [[Prototype]] этого объекта записывается объект `Employee.prototype`.
3. Оператор `new` вызывает функцию-конструктор с тремя параметрами: `this` (указывает на только что созданный объект), `name` и `salary`.

4. В теле функции `Employee` устанавливаются свойства объекта, для чего используется параметр `this`.
5. Конструктор возвращает управление, значением оператора `new` является полностью инициализированный к этому моменту объект.
6. Переменная `harry` инициализируется ссылкой на объект. Результат показан на рис. 4.3.

На рис. 4.3 видно, что у объекта `Employee.prototype` имеется прототип `Object.prototype`, который привносит метод `toString` и еще несколько методов.

Благодаря всей этой магии оператор `new` выглядит точь-в-точь как вызов конструктора в Java, C# или C++. Однако `Employee` – не класс, а просто функция. Но тогда что же такое класс? Согласно определению из учебника, классом называется множество объектов, обладающих одинаковым поведением, которое определяется методами. Все объекты, полученные путем вызова `new Employee(...)`, имеют один и тот же набор методов. В JavaScript функции-конструкторы – эквивалент классов в языках программирования на основе классов.

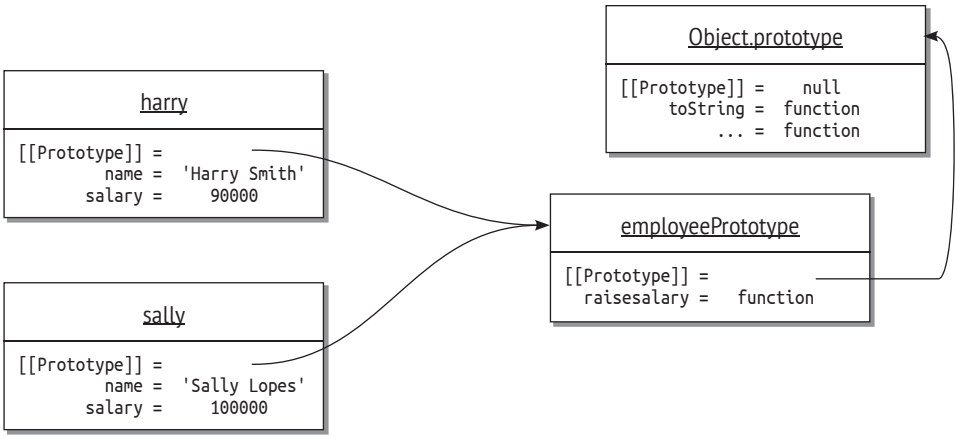

Рис. 4.3 ❖ Объекты, созданные конструктором

Вам нечасто придется задумываться о различии между традиционными классами и системой JavaScript, основанной на прототипах. В следующем разделе мы увидим, что синтаксис современного JavaScript старается придерживаться соглашений, принятых в языках на основе классов. Однако время от времени стоит напоминать себе, что класс JavaScript – всего лишь функция-конструктор, а общность поведения достигается благодаря прототипам.

4.4. Синтаксис классов

В настоящее время JavaScript обзавелся синтаксисом классов, упаковывающим функцию-конструктор и методы прототипа в знакомую форму. Вот как пример из предыдущего раздела записывается с применением синтаксиса классов:

```
class Employee {
  constructor(name, salary) {
    this.name = name
    this.salary = salary
  }
  raiseSalary(percent) {
    this.salary *= 1 + percent / 100
  }
}
```

Этот код делает *в точности* то же самое, что код из предыдущего раздела. Но собственно класса по-прежнему не существует. За кулисами объявление `class` просто объявляет функцию-конструктор `Employee`. Ключевое слово `constructor` служит для объявления тела функции-конструктора `Employee`. Метод `raiseSalary` добавляется в `Employee.prototype`. Как и в предыдущем разделе, мы конструируем объект, вызывая функцию-конструктор с помощью оператора `new`:

```
const harry = new Employee('Harry Smith', 90000)
```

> **Примечание.** Как отмечалось в предыдущих разделах, функция `constructor` не должна возвращать значения. Но даже если она все-таки что-то возвращает, то это значение игнорируется, а выражение `new` все равно возвращает вновь созданный объект.

Настоятельно рекомендую использовать синтаксис `class` (это золотое правило 4 из предисловия). Этот синтаксис берет на себя ряд канительных деталей, которыми вряд ли стоит заниматься самостоятельно. Просто помните, что `class` в JavaScript – синтаксический сахар, обертывающий функцию-конструктор и объект-прототип, в котором хранятся методы.

> **Примечание.** У класса может быть не более одного конструктора. Если вы объявите класс без конструктора, то в него автоматически будет добавлен конструктор с пустым телом.

> **Предостережение.** В отличие от объектного литерала, в объявлении класса не нужны запятые между объявлениями методов.

> **Примечание.** Классы, в отличие от функций, не поднимаются наверх. Необходимо объявить класс, прежде чем можно будет сконструировать его экземпляр.

> **Примечание.** Тело класса выполняется в строгом режиме.

4.5. Акцессоры чтения и записи

Акцессором чтения (getter) называется метод без параметров, объявленный с помощью ключевого слова `get`:

```
class Person {
  constructor(last, first) {
    this.last = last;
```

```
    this.first = first
  }
  get fullName() { return `${this.last}, ${this.first}` }
}
```

При вызове акцессора чтения скобки не ставятся, как если бы мы обращались к значению свойства:

```
const harry = new Person('Smith', 'Harry')
const harrysName = harry.fullName // 'Smith, Harry'
```

У объекта `harry` нет свойства `fullName`, но вызывается акцессор чтения. Можно считать, что акцессор чтения – это динамически вычисляемое свойство.

Можно также задать *акцессор записи* (setter) – метод с одним параметром:

```
class Person {
  ...
  set fullName(value) {
    const parts = value.split(/,\s*/)
    this.last = parts[0]
    this.first = parts[1]
  }
}
```

Акцессор записи вызывается, когда производится присваивание `fullName`:

```
harry.fullName = 'Smith, Harold'
```

У пользователей класса, обладающего акцессорами чтения и записи, создается иллюзия, что они работают со свойствами, однако автор может контролировать значения свойств и любые попытки их изменения.

4.6. Поля экземпляра и закрытые методы

Мы можем динамически создать свойство объекта в конструкторе или любом другом методе, присвоив значение `this.`*`имяСвойства`*. Такие свойства аналогичны полям экземпляра в языках, основанных на классах.

```
class BankAccount {
  constructor() { this.balance = 0 }
  deposit(amount) { this.balance += amount }
  ...
}
```

В начале 2020 года на третьей стадии рассмотрения находилось три предложения альтернативной нотации. Можно перечислить имена и начальные значения полей в объявлении класса:

```
class BankAccount {
  balance = 0
```

```
  deposit(amount) { this.balance += amount }
  ...
}
```

Поле считается *закрытым* (т. е. доступным только из методов класса), если его имя начинается знаком #:

```
class BankAccount {
  #balance = 0
  deposit(amount) { this.#balance += amount }
  ...
}
```

Метод считается закрытым, если его имя начинается знаком #.

4.7. Статические методы и поля

В объявлении class метод можно объявить с ключевым словом static. Такой метод не вызывается от имени объекта. Это обычная функция, являющаяся свойством класса, например:

```
class BankAccount {
  ...
  static percentOf(amount, rate) { return amount * rate / 100 }
  ...
  addInterest(rate) {
    this.balance += BankAccount.percentOf(this.balance, rate)
  }
}
```

Для вызова статического метода, все равно – изнутри или извне класса, нужно добавить имя класса, как в примере выше.

На внутреннем уровне статический метод является свойством конструктора. В стародавние времена это нужно было делать вручную:

```
BankAccount.percentOf = function(amount, rate) {
  return amount * rate / 100
}
```

Таким же образом можно определить эквивалент статических полей:

```
BankAccount.OVERDRAFT_FEE = 30
```

В начале 2020 года синтаксис статических полей класса находился на третьей стадии рассмотрения.

```
class BankAccount {
  static OVERDRAFT_FEE = 30
  ...
  withdraw(amount) {
    if (this.balance < amount) {
```

```
      this.balance -= BankAccount.OVERDRAFT_FEE
    }
    ...
  }
}
```

Статическое поле просто становится свойством функции-конструктора. Как и в случае статических методов, для доступа к такому полю нужно указать имя класса: `BankAccount.OVERDRAFT_FEE`.

Предложение о закрытых статических полях и методах (с префиксом #) в настоящее время также находится на третьей стадии рассмотрения.

Акцессоры чтения и записи можно объявлять статическими методами. Как всегда, акцессор записи может выполнять проверку ошибок:

```
class BankAccount {
  ...
  static get OVERDRAFT_FEE() {
    return this.#OVERDRAFT_FEE // в статическом методе this - это функция-конструктор
  }
  static set OVERDRAFT_FEE(newValue) {
    if (newValue > this.#OVERDRAFT_FEE) {
      this.#OVERDRAFT_FEE = newValue
    }
  }
}
```

4.8. Подклассы

Ключевой концепцией объектно-ориентированного программирования является наследование. Класс определяет поведение своих экземпляров. Мы можем создать подкласс данного класса (который называется суперклассом), экземпляры которого будут вести себя в каком-то отношении по-другому, но остальное поведение унаследуют от суперкласса.

Стандартный учебный пример – иерархия наследования с суперклассом `Employee` и подклассом `Manager`. Ожидается, что работники получают за свою работу зарплату, а менеджеры, помимо основной зарплаты, получают бонусы, если достигают поставленных целей.

В JavaScript, как и в Java, для выражения такой связи между классами `Employee` и `Manager` служит ключевое слово `extends`:

```
class Employee {
  constructor(name, salary) { ... }
  raiseSalary(percent) { ... }
  ...
}

class Manager extends Employee {
  getSalary() { return this.salary + this.bonus }
  ...
}
```

За кулисами организуется цепочка прототипов (см. рис. 4.4). Прототипом `Manager.prototype` становится `Employee.prototype`. Таким образом, любой метод, не объявленный в подклассе, ищется в его суперклассе.

Например, мы можем вызвать метод `raiseSalary` объекта менеджера:

```
const boss = new Manager(...)
boss.raiseSalary(10) // вызывается Employee.prototype.raiseSalary
```

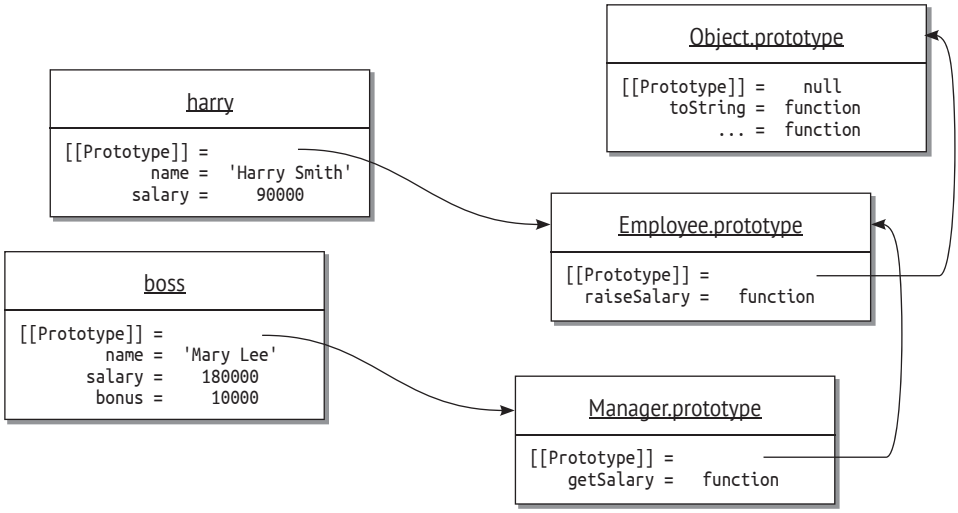

Рис. 4.4 ❖ Цепочка прототипов для наследования

До появления синтаксиса `extends` программисты на JavaScript должны были организовывать такие цепочки прототипов самостоятельно.

Оператор `instanceof` проверяет, принадлежит ли объект классу или одному из его подклассов. Технически оператор обходит цепочку прототипов объекта и проверяет, есть ли в ней прототип данной функции-конструктора. Например, выражение

```
boss instanceof Employee
```

равно `true`, поскольку `Employee.prototype` встречается в цепочке прототипов `boss`.

> **Примечание.** В Java ключевое слово `extends` применяется для расширения фиксированного класса. В JavaScript механизм `extends` более динамичный. В правой части `extends` может находиться любое выражение, которое дает функцию (или `null`, если нужно породить класс, не расширяющий `Object`). В разделе 4.11 «Классовые выражения» приведен пример.

> **Примечание.** В Java и C++ часто определяют абстрактные классы или интерфейсы специально для того, чтобы можно было вызывать методы, определенные в подклассах. В JavaScript правильность применения методов не проверяется на этапе компиляции, поэтому нужды в абстрактных методах нет. Например, предположим, что моделируются штатные работники и работники, нанятые по срочному договору,

и требуется рассчитывать зарплаты для объектов обоих классов. В статически типизированном языке мы ввели бы суперкласс или интерфейс `Salaried` с абстрактным методом `getSalary`. В JavaScript нужно просто вызвать `person.getSalary()`.

4.9. Переопределение методов

Предположим, что суперкласс и подкласс имеют метод `getSalary`:

```
class Employee {
  ...
  getSalary() { return this.salary }
}

class Manager extends Employee {
  ...
  getSalary() { return this.salary + this.bonus }
}
```

Теперь рассмотрим вызов метода:

```
const empl = ...
const salary = empl.getSalary()
```

Если `empl` – ссылка на трудягу-работника, то вызывается метод `Employee.prototype.getSalary`. Если же `empl` ссылается на менеджера, то вызывается метод `Manager.prototype.getSalary`. Это явление – когда вызванный метод зависит от того, на какой объект указывает ссылка, – называется *полиморфизмом*. В JavaScript полиморфизм является простым следствием поиска в цепочке прототипов.

В этой ситуации говорят, что метод `getSalary` в классе `Manager` *переопределяет* метод `getSalary` в классе `Employee`.

Иногда мы хотим вызвать метод суперкласса из подкласса, например:

```
class Manager extends Employee {
  ...
  getSalary() { return super.getSalary() + this.bonus }
}
```

Здесь `super` начинает поиск с родителя объекта-прототипа, в котором был объявлен данный метод. В нашем примере вызов `super.getSalary` не заглядывает в `Manager.prototype`, поскольку иначе возникла бы бесконечная рекурсия. Вместо этого вызывается метод `getSalary` из `Employee.prototype`.

> **Примечание.** В этом разделе для демонстрации переопределения методов мы использовали метод `getSalary`. Можно переопределять также акцессоры чтения и записи:
>
> ```
> class Manager extends Employee {
>   ...
>   get salary() { return super.salary + this.bonus }
> }
> ```

4.10. Конструирование подкласса

В конструкторе подкласса *обязательно* вызывать конструктор суперкласса. Для этого используется синтаксис `super(...)`, как в Java. Внутри скобок поместите аргументы, которые нужно передать конструктору суперкласса.

```
class Manager extends Employee {
  constructor(name, salary, bonus) {
    super(name, salary) // вызывать конструктор суперкласса обязательно
    this.bonus = bonus // потом допустим такой код
  }
  ...
}
```

Ссылку `this` разрешается использовать только после вызова `super`. Однако если вы не предоставили конструктор подкласса, то он будет сгенерирован автоматически. И этот автоматический конструктор передает все аргументы конструктору суперкласса. (Это гораздо полезнее, чем в Java или C++, где вызывается конструктор суперкласса без аргументов.)

```
class Manager extends Employee {
  // Нет конструктора
  getSalary() { ... }
}

const boss = new Manager('Mary Lee', 180000) // вызывается Employee('Mary Lee', 180000)
```

До того как в JavaScript были добавлены ключевые слова `extends` и `super`, реализовать конструктор подкласса, который вызывал бы конструктор суперкласса, было отнюдь не тривиально. Этот процесс – теперь уже ненужный – требует продвинутых средств, которые описаны в главе 11.

> **Примечание.** Как вы знаете, в JavaScript нет настоящих классов. Класс – это просто функция-конструктор. А подкласс – это функция-конструктор, которая вызывает конструктор суперкласса.

4.11. Классовые выражения

Мы можем объявлять анонимные классы – точно так же, как анонимные функции:

```
const Employee = class {
  constructor(name, salary) {
    this.name = name
    this.salary = salary
  }
  raiseSalary(percent) {
    this.salary *= 1 + percent / 100
  }
}
```

Напомним, что объявление class возвращает функцию-конструктор. Теперь эта функция хранится в переменной Employee. В этом примере нет никакого выигрыша по сравнению с нотацией именованных классов class Employee { . . . }.

Но вот более полезное приложение. Можно написать методы, которые «подмешивают» некоторую функциональность в существующий класс:

```
const withToString = base =>
  class extends base {
    toString() {
      let result = '{'
      for (const key in this) {
        if (result !== '{') result += ', '
        result += `${key}=${this[key]}`
      }
      return result + '}'
    }
  }
```

Вызовем эту функцию, передав ей класс (т. е. функцию-конструктор), чтобы получить пополненный класс:

```
const PrettyPrintingEmployee = withToString(Employee) // новый класс
e = new PrettyPrintingEmployee('Harry Smith', 90000) // экземпляр нового класса
console.log(e.toString())
// Печатается {name=Harry Smith, salary=90000}, а не [object Object]
```

4.12. Ссылка this

В этом разделе, помеченном значком Безумного шляпника, мы более внимательно рассмотрим ссылку this. Можете пропустить данный раздел, если используете this только в конструкторах, методах и стрелочных функциях, но не внутри именованных функций.

Чтобы понять, почему this может стать причиной бед, рассмотрим сначала оператор new. Что случится, если вызвать функцию-конструктор без new? Если написать код

```
let e = Employee('Harry Smith', 90000) // забыли new
```

в строгом режиме, то переменной this будет присвоено значение undefined.

К счастью, эта проблема возникает только в старомодных объявлениях функции-конструктора. Если использовать синтаксис class, то вызывать конструктор без new запрещается.

> **Предостережение.** Если не пользоваться синтаксисом class, то можно объявлять функции-конструкторы, так что они смогут работать и с new, и без new. Примером может служить функция Number:
>
> ```
> const price = Number('19.95')
>   // Разбирает строку и возвращает примитивное число, а не объект
> ```

```
const aZeroUnlikeAnyOther = new Number(0)
  // Конструирует новый объект
```

Вызов конструктора без new – редкость в современном JavaScript.

А вот еще одна потенциальная проблема. Можно вызвать метод, не указав объекта. В таком случае this будет равно undefined:

```
const doLater = (what, arg) => { setTimeout(() => what(arg), 1000) }
doLater(BankAccount.prototype.deposit, 500) // ошибка
```

Когда выражение what(arg) вычисляется спустя одну секунду, вызывается метод deposit. В этом методе происходит ошибка при попытке доступа к this.balance, потому что this равно undefined.

Если мы хотим пополнить конкретный счет, нужно просто указать этот счет:

```
doLater(amount => harrysAccount.deposit(amount), 500)
```

Далее рассмотрим вложенные функции. Внутри вложенной функции, объявленной с ключевым словом function, ссылка this равна undefined. Попытавшись использовать this в функции обратного вызова, вы испытаете горькое разочарование:

```
class BankAccount {
  ...
  spreadTheWealth(accounts) {
    accounts.forEach(function(account) {
      account.deposit(this.balance / accounts.length)
        // Ошибка - this равна undefined внутри вложенной функции
    })
    this.balance = 0
  }
}
```

Здесь this.balance *не* ссылается на остаток банковского счета. Ссылка this равна undefined, поскольку встретилась во вложенной функции.

Исправить это проще всего, воспользовавшись стрелочной функцией в качестве обратного вызова:

```
class BankAccount {
  ...
  spreadTheWealth(accounts) {
    accounts.forEach(account => {
      account.deposit(this.balance / accounts.length) // this привязана правильно
    })
    this.balance = 0
  }
}
```

В стрелочной функции this статически привязывается к тому же, что this означает вне стрелочной функции, – в данном случае к объекту BankAccount, от имени которого вызван метод spreadTheWealth.

Примечание. До появления стрелочных функций программисты на JavaScript использовали обходной маневр – инициализировали другую переменную ссылкой `this`:

```
spreadTheWealth(accounts) {
  const that = this
  accounts.forEach(function(account) {
    account.deposit(that.balance / accounts.length)
  })
  this.balance = 0
}
```

Вот еще один неочевидный пример. Любой вызов метода `obj.method(args)` можно записать также в виде `obj['method'](args)`. По этой причине `this` устанавливается в `obj`, если выполнить вызов `obj[index](args)`, где `obj[index]` – функция, даже если никакого оператора точка нет и в помине.

Сконструируем такую ситуацию на примере массива обратных вызовов:

```
class BankAccount {
  constructor() {
    this.balance = 0
    this.observers = []
  }
  addObserver(f) {
    this.observers.push(f)
  }
  notifyObservers() {
    for (let i = 0; i < this.observers.length; i++) {
      this.observers[i]()
    }
  }
  deposit(amount) {
    this.balance += amount
    this.notifyObservers()
  }
  ...
}
```

Пусть теперь имеется банковский счет:

```
const acct = new BankAccount()
```

Добавим наблюдателя:

```
class UserInterface {
  log(message) {
    ...
  }
  start() {
    acct.addObserver(function() { this.log('Еще денег!') })
    acct.deposit(1000)
  }
}
```

Чему равна ссылка this, когда вызывается функция, переданная addObserver? *Массиву наблюдателей!* Именно так она была установлена в вызове this.observers[i](). Поскольку у массива нет метода log, возникает ошибка во время выполнения. И снова для исправления нужно использовать стрелочную функцию:

```
acct.addObserver(() => { this.log('Еще денег!') })
```

> **Совет.** Динамическая установка this, подчиняющаяся запутанному набору правил, – источник всяческих проблем. Чтобы избежать неприятностей, не используйте this внутри функций, определенных с помощью ключевого слова function. Безопасно использовать this в методах и конструкторах, а также в стрелочных функциях, определенных внутри методов и конструкторов. В этом заключается золотое правило 5.

Упражнения

1. Напишите функцию createPoint, которая создает точку с координатами *x* и *y* на плоскости. Предоставьте методы getX, getY, translate и scale. Метод translate производит параллельный перенос точки на величину, заданную параметрами *x* и *y*. Метод scale умножает обе координаты на заданный коэффициент. Пользуйтесь только приемами, описанными в разделе 4.1.
2. Повторите предыдущее упражнение, но теперь реализуйте функцию-конструктор и используйте прототипы, как описано в разделе 4.2.
3. Повторите предыдущее упражнение с использованием синтаксиса классов.
4. Повторите предыдущее упражнение, но предоставьте акцессоры чтения и записи для координат *x* и *y*. В акцессоре записи проверяйте, что аргумент – число.
5. Рассмотрим следующую функцию, которая делает строку «способной на приветствие», добавляя метод greet:

   ```
   function createGreetable(str) {
     const result = new String(str)
     result.greet = function(greeting) { return `${greeting}, ${this}!` }
     return result
   }
   ```

   Типичное использование:

   ```
   const g = createGreetable('World')
   console.log(g.greet('Hello'))
   ```

   У этой функции есть недостаток: в каждой такой строке имеется собственная копия метода greet. Сделайте так, чтобы createGreetable возвращала объект, прототип которого содержит метод greet. Убедитесь, что по-прежнему можете вызывать все методы строки.

6. Напишите метод `withGreeter`, который добавляет метод `greet` к любому классу, порождая новый класс:

```
const GreetableEmployee = withGreeter(Employee)
const e = new GreetableEmployee('Harry Smith', 90000)
console.log(e.greet('Hello'))
```

   Указание: см. раздел 4.11 «Классовые выражения».
7. Перепишите класс `Employee`, используя закрытые поля экземпляра, как показано в разделе 4.6.
8. Классический пример абстрактного класса – узел дерева. Узлы бывают двух видов: имеющие потомков (родители) и без потомков (листья).

```
class Node {
  depth() { throw Error("abstract method") }
}
class Parent extends Node {
  constructor(value, children) { ... }
  depth() { return 1 + Math.max(...children.map(n => n.depth())) }
}
class Leaf extends Node {
  constructor(value) { ... }
  depth() { return 1 }
}
```

   Так мы стали бы моделировать узлы дерева в Java или C++. Но в JavaScript не нужен абстрактный класс для вызова `n.depth()`. Перепишите эти классы без использования наследования и добавьте тестовую программу.
9. Напишите класс `Random` со статическими методами

```
Random.nextDouble(low, high)
Random.nextInt(low, high)
Random.nextElement(array)
```

   которые возвращают случайное число в диапазоне от `low` (включая) до `high` (не включая) или случайный элемент заданного массива.
10. Напишите класс `BankAccount` и его подклассы `SavingsAccount` и `CheckingAccount`. В классе `SavingsAccount` (сберегательный счет) должно быть поле экземпляра, содержащее процентную ставку, и метод `addInterest` для прибавления процентов. Класс `CheckingAccount` (текущий счет) взимает плату за каждое снятие денежных средств. Не изменяйте напрямую состояние суперкласса, пользуйтесь его методами.
11. Нарисуйте диаграмму объектов `SavingsAccount` и `CheckingAccount` из предыдущего упражнения по образцу на рис. 4.4.
12. Гарри пытается с помощью следующего кода изменить класс CSS по нажатии кнопки:

```
const button = document.getElementById('button1')
button.addEventListener('click', function () {
  this.classList.toggle('clicked')
})
```

Но код не работает. Почему?
Салли, поискав в интернете, предложила такое решение:

```
button.addEventListener('click', event => {
  event.target.classList.toggle('clicked')
})
```

Оно работает, но Гарри кажется, что здесь какой-то обман. Что, если бы прослушиватель не сделал кнопку доступной с помощью свойства `event.target`? Исправьте этот код, отказавшись от использования `this` и параметра `event`.

13. В разделе 4.12 мы видели, что такой код не работает:

```
const action = BankAccount.prototype.deposit
action(1000)
```

Можно ли заставить его работать, получив метод `action` от экземпляра:

```
const harrysAccount = new BankAccount()
const action = harrysAccount.deposit
action(1000)
```

Объясните свой ответ.

14. В предыдущем упражнении мы определили функцию `action`, которая кладет денежные средства на счет `harrysAccount`. Пока особого смысла в этом не видно, поэтому добавим контекст. Показанная ниже функция вызывает переданную ей функцию с задержкой и передает ей величину задержки в качестве аргумента.

```
function invokeLater(f, delay) {
  setTimeout(() => f(delay), delay)
}
```

Она позволяет Гарри заработать 1000 долларов спустя 1000 миллисекунд:

```
invokeLater(amount => harrysAccount.deposit(amount), 1000)
```

Но как быть с Салли? Напишите общую функцию `depositInto`, которую можно было бы вызвать следующим образом:

```
invokeLater(depositInto(sallysAccount), 1000)
```

Глава 5

Числа и даты

В этой короткой главе мы рассмотрим JavaScript API для работы с числами и большими целыми. А затем обратимся к операциям с датами. Мы увидим, что в JavaScript дату можно преобразовать в число – количество миллисекунд. Это преобразование не особенно полезно, но оправдывает включение обеих тем в одну главу вместо двух, еще более коротких.

5.1. Числовые литералы

Все числа в JavaScript – с плавающей точкой «двойной точности» в смысле, определенном в стандарте IEEE 754. В двоичном представлении они занимают 8 байт.

Целочисленные литералы можно записывать в десятичной, шестнадцатеричной, восьмеричной и двоичной форме:

```
42
0x2A
0o52
0b101010
```

> **Примечание.** Архаичная запись в восьмеричной форме с начальным нулем и без `o` (например, `052`) в строгом режиме запрещена.

Для записи литералов с плавающей точкой можно использовать экспоненциальную нотацию:

```
4.2e-3
```

Буквы `e` `x` `o` `b` могут быть как строчными, так и заглавными: формы `4.2E-3` и `0X2A` допустимы.

> **Примечание.** В C++ и Java допустимы шестнадцатеричные литералы с плавающей точкой, например `0x1.0p-10` = 2^{-10} = `0.0009765625`. Такая нотация в JavaScript не поддерживается.

Предложение разрешить использование знака подчеркивания в числовых литерах в 2020 году находится на третьей стадии рассмотрения. Подчерки можно помещать между любыми цифрами, чтобы сделать число удобочитаемым. Но предназначены они только для человека, а на этапе разбора числа удаляются. Например:

```
const speedOfLight = 299_792_458 // то же, что 299792458
```

Глобальные переменные `Infinity` и `NaN` обозначают «бесконечность» и «не число». Например, `1 / 0` равно `Infinity`, а `0 / 0` равно `NaN`.

5.2. Форматирование чисел

Для форматирования целого числа в системе счисления с основанием от 2 до 36 используется метод `toString`:

```
const n = 3735928559
n.toString(16) // 'deadbeef'
n.toString(8) // '33653337357'
n.toString(2) // '11011110101011011011111011101111'
```

Числа с плавающей точкой также можно представить в системе счисления с основанием, отличным от 10:

```
const almostPi = 3.14
almostPi.toString(16) // 3.23d70a3d70a3e
```

Метод `toFixed` форматирует число с плавающей точкой в фиксированном формате с заданным числом цифр после десятичной точки. Вызов `x.toExponential(p)` служит для представления числа в экспоненциальном формате с одной цифрой до и `p – 1` цифрами после десятичной точки. А вызов `x.toPrecision(p)` возвращает число с `p` значащими цифрами:

```
const x = 1 / 600 // 0.0016666666666666668
x.toFixed(4) // '0.0017'
x.toExponential(4) // '1.667e-3'
x.toPrecision(4) // '0.001667'
```

Метод `toPrecision` переключается на экспоненциальный формат, если в противном случае число значащих цифр или нулей оказалось бы слишком велико (см. упражнение 3).

> **Примечание.** В стандартной библиотеке JavaScript нет эквивалента функции C `printf`, но существуют сторонние реализации, например https://github.com/alexei/sprintf.js.
> Метод `console.log` поддерживает спецификаторы в стиле `printf`: `%d`, `%f`, `%s`, но не модификаторы ширины, заполнения и точности.

5.3. Разбор чисел

В главе 1 мы видели, как разбирать строки, содержащие числа:

```
const notQuitePi = parseFloat('3.14') // число 3.14
const evenLessPi = parseInt('3') // целое число 3
```

Эти функции игнорируют пробельные префиксы и нечисловые суффиксы. Например, `parseInt(' 3A')`также равно `3`.

Результат равен `NaN`, если после необязательных пробельных символов нет ни одной цифры. Например, `parseInt(' A3')` равно `NaN`.

Функция `parseInt` принимает число в шестнадцатеричной нотации: `parseInt('0x3A')` равно `58`.

Иногда нужно, чтобы строка содержала только десятичные цифры, без всяких начальных пробелов и суффиксов. В таком случае лучше использовать регулярное выражение:

```
const intRegex = /^[+-]?[0-9]+$/
if (intRegex.test(str)) value = parseInt(str)
```

Для чисел с плавающей точкой регулярное выражение более сложное:

```
const floatRegex = /^[+-]?((0|[1-9][0-9]*)(\.[0-9]*)?|\.[0-9]+)([eE][+-]?[0-9]+)?$/
if (floatRegex.test(str)) value = parseFloat(str)
```

Подробнее о регулярных выражениях см. главу 6.

> **Предостережение.** Интернет кишит почти правильными рецептами для распознавания строк, представляющих числа в JavaScript, но дьявол кроется в деталях. Приведенные выше регулярные выражения принимают те и только те десятичные числовые литералы, которые допускаются стандартом JavaScript, им может предшествовать необязательный знак. Но внутренних знаков подчеркивания (например, 1_000_000) они не поддерживают.

Для разбора чисел в системе счисления с основанием, отличным от 10, укажите основание от 2 до 36 во втором аргументе.

```
parseInt('deadbeef', 16) // 3735928559
```

5.4. Функции и константы в классе Number

Функции `Number.parseInt` и `Number.parseFloat` совпадают с глобальными функциями `parseInt` и `parseFloat`.

Вызов `Number.isNaN(x)` проверяет, равно ли `x` специальному значению `NaN` (не число). (Проверка `x === NaN` не годится, потому что никакие два значения `NaN` не равны друг другу.)

Чтобы проверить, что значение `x` отличается от `Infinity`, `-Infinity` и `NaN`, вызовите функцию `Number.isFinite(x)`.

> **Предостережение.** Не используйте глобальные функции `isNaN` и `isFinite` – они сначала преобразуют нечисловой аргумент в число, результат получается бесполезным:
>
> ```
> isNaN('Hello') // true
> isFinite([0]) // true
> ```

Статические методы `Number.isInteger` и `Number.isSafeInteger` проверяют, является ли аргумент соответственно целым числом и числом в безопасном

диапазоне, в котором не происходит округления. Безопасный диапазон простирается от `Number.MIN_SAFE_INTEGER` ($-2^{53} + 1$, или −9 007 199 254 740 991) до `Number.MAX_SAFE_INTEGER` ($2^{53} - 1$, или 9 007 199 254 740 991).

Наибольшее представимое число равно `Number.MAX_VALUE` ($(2 - 2^{-52}) \times 2^{1023}$, приблизительно 1.8×10^{308}). Наименьшее представимое число равно `Number.MIN_VALUE` (2^{-1074}, приблизительно 5×10^{-324}). `Number.EPSILON` (2^{-52}, приблизительно 2.2×10^{-16}) – это промежуток между 1 и следующим представимым числом, большим 1.

Наконец, `Number.NaN`, `Number.POSITIVE_INFINITY` и `Number.NEGATIVE_INFINITY` совпадают с глобальными `NaN`, `Infinity` и `-Infinity` соответственно. Можете использовать их, если опасаетесь, что кто-то мог определить локальные переменные с именами `NaN` и `Infinity`.

В табл. 5.1 приведены наиболее полезные средства класса `Number`.

Таблица 5.1. Полезные функции, методы и константы из класса `Number`

Имя	Описание
Функции	
`isNaN(x)`	`true`, если `x` равно `NaN`. Отметим, что использовать оператор `===` нельзя, т. к. `x === NaN` всегда `false`
`isFinite(x)`	`true`, если `x` не равно `±Infinity`, `NaN`
`isSafeInteger(x)`	`true`, если `x` – целое число в «безопасном» диапазоне, определенном выше
Методы	
`toString(base)`	Число в системе счисления с основанием `base` (от 2 до 36). `(200).toString(16)` равно `'c8'`
`toFixed(digitsAfterDecimalPoint)`, `toExponential(significantDigits)`, `toPrecision(significantDigits)`	Число в формате с фиксированной точкой, или в экспоненциальном формате, или в том из этих двух, который удобнее. Форматирование `0.001666` с четырьмя цифрами дает `'0.0017'`, `'1.667e-3'`, `'0.001667'`
Константы	
`MIN_SAFE_INTEGER`, `MAX_SAFE_INTEGER`	Безопасный диапазон целых чисел, которые можно представить числами с плавающей точкой без округления
`MIN_VALUE`, `MAX_VALUE`	Диапазон всех чисел с плавающей точкой

5.5. Математические функции и константы

В классе `Math` определены функции и константы для математических вычислений: логарифмы, тригонометрические функции и т. п. В табл. 5.2 приведен полный перечень. Большинство функций специальные, но есть несколько представляющих общий интерес.

Функции `max` и `min` возвращают наибольший и наименьший аргументы, количество аргументов произвольно:

```
Math.max(x, y) // большее из x и y
Math.min(...values) // наименьший элемент массива values
```

Функция `Math.round` производит округление до ближайшего целого. Положительные числа, дробная часть которых ⩾ 0.5, и отрицательные числа с дробной частью > 0.5 округляются с избытком.

Функция `Math.trunc` просто отбрасывает дробную часть.

```
Math.round(2.5) // 3
Math.round(-2.5) // -2
Math.trunc(2.5) // 2
```

Вызов `Math.random()` возвращает число с плавающей точкой от 0 (включая) до 1 (не включая). Для получения случайного числа с плавающей точкой или целого числа в диапазоне от `a` (включая) до `b` (не включая) нужно вызвать соответственно

```
const randomDouble = a + (b - a) * Math.random()
const randomInt = a + Math.trunc((b - a) * Math.random()) // где a, b целые
```

Таблица 5.2. Функции и константы в классе Math

Имя	Описание
Функции	
`min(values...)`, `max(values)`	Эти функции могут иметь произвольное число аргументов
`abs(x)`, `sign(x)`	Абсолютная величина и знак (1, 0, –1)
`random()`	Случайное число $0 \leqslant r < 1$
`round(x)`, `trunc(x)`, `floor(x)`, `ceil(x)`	Округление до ближайшего целого, отбрасывание дробной части и округление до предыдущего меньшего или следующего большего целого
`fround(x)`, `ftrunc(x)`, `floor(x)`, `fceil(x)`	Округление до 32-разрядного числа с плавающей точкой
`pow(x, y)`, `exp(x)`, `expm1(x)`, `log(x)`,`log2(x)`, `log10(x)`, `log1p(x)`	x^y, e^x, $e^x - 1$, $\ln(x)$, $\log_2(x)$, $\log_{10}(x)$, $\ln(1 + x)$
`sqrt(x)`, `cbrt(x)`, `hypot(x, y)`	$\sqrt{x}$, $\sqrt[3]{x}$, $\sqrt{x^2 + y^2}$
`sin(x)`, `cos(x)`, `tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`, `atan2(y, x)`	Тригонометрические функции
`sinh(x)`, `cosh(x)`, `tanh(x)`, `asinh(x)`, `acosh(x)`, `atanh`	Гиперболические функции
Константы	
`E`, `PI`, `SQRT2`, `SQRT1_2`, `LN2`, `LN10`,`LOG2E`, `LOG10E`	e, π, $\sqrt{2}$, $\sqrt{1/2}$, $\ln(2)$, $\ln(10)$, $\log_2(e)$, $\log_{10}(e)$

5.6. Большие целые

Большим целым называется целое число с произвольным количеством знаков. Большой целый литерал имеет суффикс `n`, например `815915283247897734345611269596115894272000000000n`. Можно также преобразовать целочисленное выражение в большое целое: `BigInt(expr)`.

Оператор `typeof`, примененный к большому целому, возвращает `'bigint'`.

Арифметические операторы, примененные к большим целым, возвращают новое большое целое:

```
let result = 815915283247897734345611269596115894272000000000n * BigInt(41)
  // Результат равен 33452526613163807108170062053440751665152000000000n
```

> **Примечание.** Операндами арифметического оператора не могут быть большое целое и значение какого-то другого типа. Например, `815915283247897734345611269596115894272000000000n * 41` – ошибка.

Оператор / для больших целых оставляет большое целое частное и отбрасывает остаток, например `100n / 3n` равно `33n`.

В классе `BigInt` всего две функции, носящие технический характер. Вызовы `BigInt.asIntN(bits, n)` и `BigInt.asUintN(bits, n)` приводят `n` по модулю 2^{bits} к интервалу $[-2^{bits-1} \ldots 2^{bits-1} - 1]$ или $[0 \ldots 2^{bits} - 1]$ соответственно.

5.7. Конструирование дат

Прежде чем приступать к изучению JavaScript API для работы с датами, напомним несколько фактов об измерении времени на нашей планете.

Исторически фундаментальная единица времени – секунда – была выведена из наблюдений за вращением Земли вокруг своей оси. Полный оборот совершается за 24 часа, или 24 × 60 × 60 = 86 400 секунд, поэтому кажется, что для точного определения секунды нужны лишь астрономические измерения. К сожалению, орбита Земли немного флуктуирует, поэтому необходимо более точное определение. В 1967 году новое определение секунды, согласующееся с историческим определением, было выведено из свойства, внутренне присущего атомам цезия-133. С тех пор для хранения официального времени используется сеть атомных часов.

Периодически официальные хранители времени синхронизируют абсолютное время с вращением Земли. Поначалу немного корректировались официальные секунды, но начиная с 1972 года стали вставлять «високосные» секунды. (Теоретически может потребоваться также удалять секунды, но до сих пор такого не случалось.) Понятно, что високосные секунды – источник всяческих проблем, и во многих компьютерных системах вместо них применяется «сглаживание», когда время искусственно замедляется или ускоряется непосредственно перед високосной секундой, так что в сутках всегда остается 86 400 секунд. Это работает, потому что локальное время компьютера далеко от абсолютной точности, так что компьютерам приходится синхронизироваться с внешней службой времени.

Поскольку людям в любой точке света хотелось бы, чтобы полночь более-менее соответствовала середине ночи, местное время различается. Но для сравнения моментов времени необходима общая точка отсчета. В силу исторических причин за такую точку взято время на меридиане, проходящем через Королевскую обсерваторию в Гринвиче (без поправки на летнее время). Это время называется координированным универсальным временем (Coordinated Universal Time), или UTC. Акроним является компромиссом

между английским и французским (Temps Universel Coordiné) написанием, но не совпадает с первыми буквами ни того, ни другого.

Для представления времени в компьютере удобно иметь фиксированную точку, от которой можно отсчитывать как вперед, так и назад. В качестве такой точки, называемой началом отсчета, или «эпохой», выбрана полночь по времени UTC в четверг 1 января 1970 года.

В JavaScript время измеряется в сглаженных миллисекундах от начала отсчета. Допустимый диапазон составляет ±100 000 000 дней.

Для представления момента времени в JavaScript используется формат ISO 8601: `YYYY-MM-DDTHH:mm:ss.sss`**`Z`**, четыре цифры для года, по две цифры для месяца, дня, часа, минуты и секунды и три цифры для миллисекунды. Дни и часы разделены буквой `T`, а суффикс Z означает нулевое смещение от UTC.

Например, начало отсчета имеет вид

```
1970-01-01T00:00:00.000Z
```

> **Примечание.** Вам, наверное, интересно, как в этот формат укладывается дата, отстоящая на 100 000 000 дней – примерно 274 000 лет – от начала отсчета. И как быть с датами «до новой эры»? Для таких дат год задается шестью цифрами со знаком: ±YYYYYY. Максимальная дата, представимая в JavaScript, равна
>
> `+275760-09-13T00:00:00.000Z`
>
> Году `0001` предшествует год `0000`, а предыдущий год записывается в виде `-000001`.

В JavaScript момент времени представляется экземпляром класса `Date`. Было бы лучше назвать этот класс `Time`, однако существующий класс унаследовал имя и ряд изъянов от класса `Date` в Java, а к ним добавил еще и собственные причуды.

Наиболее полезные средства класса `Date` приведены в табл. 5.3.

Таблица 5.3. Полезные конструкторы, функции и методы класса `Date`

Имя	Описание
Конструкторы	
`new Date(iso8601String)`	Конструирует объект `Date` из строки в формате ISO 8601, например `'1969-07-20T20:17:40.000Z'`
`new Date()`	Конструирует объект `Date`, представляющий текущее время
`new Date(millisecondsFromEpoch)`	
`new Date(year, zeroBasedMonth, day, hours, minutes, seconds, milliseconds)`	Используется *местный часовой пояс*. Необходимо по крайней мере два аргумента
Функции	
`UTC(year, zeroBasedMonth, day, hours, minutes, seconds, milliseconds)`	Возвращает количество миллисекунд от начала отсчета, а не объект `Date`
Методы	
`getUTCFullYear()`, `getUTCMonth()`, `getUTCDate()`, `getUTCHours()`, `getUTCMinutes()`, `getUTCSeconds()`, `getUTCMilliseconds()`	Месяц принимает значения от 0 до 11, день (date) – от 1 до 31, час – от 0 до 23

Таблица 5.3 *(окончание)*

Имя	Описание
`getUTCDay()`	День недели, от 0 (воскресенье) до 6 (суббота)
`getTime()`	Количество миллисекунд от начала отсчета
`toISOString()`	Строка в формате ISO 8601, например `'1969-07-20T20:17:40.000Z'`
`toLocaleString(locale, options)`, `toLocaleDateString(locale, options)`, `toLocaleTimeString(locale, options)`	Дата и время в понятном человеку формате, только дата, только время. Сведения о локалях и описание всех параметров см. в главе 8

Дату можно сконструировать из строки в формате ISO 8601 или задав количество миллисекунд от начала отсчета:

```
const epoch = new Date('1970-01-01T00:00:00.000Z')
const oneYearLater = new Date(365 * 86400 * 1000) // 1971-01-01T00:00:00.000Z
```

Конструктор `Date` без аргументов создает объект, представляющий текущее время.

```
const now = new Date()
```

> **Предостережение.** Не вызывайте функцию `Date` без `new`. При таком вызове игнорируются все аргументы и возвращается не объект `Date`, а строка, описывающая текущее время, – и даже не в формате ISO 8601:
>
> ```
> Date(365 * 86400 * 1000)
>   // аргумент игнорируется, возвращается строка
>   // 'Mon Jun 24 2020 07:23:10 GMT+0200 (Central European Summer Time)'
> ```
>
> **Предостережение.** Если объекты `Date` встречаются в арифметических выражениях, то они автоматически преобразуются либо в строковый формат, описанный в предыдущем предостережении, либо в количество миллисекунд от начала отсчета:
>
> ```
> oneYearLater + 1
>   // 'Fri Jan 01 1971 01:00:00 GMT+0100 (Central European Summer Time)1'
> oneYearLater * 1 // 31536000000
> ```
>
> Это полезно только для вычисления интервала между двумя датами:
>
> ```
> const before = new Date()
> // Что-то сделать
> const after = new Date()
> const millisecondsElapsed = after - before
> ```

Можно сконструировать объект `Date`, представляющий время в *местном часовом поясе*:

```
new Date(year, zeroBasedMonth, day, hours, minutes, seconds, milliseconds)
```

Все аргументы, начиная с `day`, необязательны. (Необходимо по крайней мере два аргумента, чтобы отличить эту форму от вызова `new Date(millisecondsFromEpoch)`.)

В силу исторических причин месяцы нумеруются с нуля, а дни – с единицы. Например, я пишу эти строки, находясь на один час к востоку от Гринвичской обсерватории. Вычисление

```
new Date(1970, 0 /* January */, 1, 0, 0, 0, 0, 0) // осторожно – местное время
```

дает

```
1969-12-30T23:00:00.000Z
```

Сделав то же самое, вы, возможно, получите другой результат – все зависит от вашего часового пояса.

> **Предостережение.** Если передать значения `zeroBasedMonth`, `day`, `hours` и т. д. за пределами допустимого диапазона, то дата скорректируется без каких-либо сообщений. Например, `new Date(2019, 13, -2)` равно January 29, 2020.

5.8. Функции и методы класса Date

В классе `Date` имеется три статические функции:

- `Date.UTC(year, zeroBasedMonth, day, hours, minutes, seconds, milliseconds)`
- `Date.parse(dateString)`
- `Date.now()`

Функция `UTC` похожа на конструктор с несколькими аргументами, но возвращает UTC-дату.

Функция `parse` разбирает строки в формате ISO 8601 и в зависимости от реализации может принимать другие форматы (см. упражнение 17).

Функция `Date.now()` возвращает текущие дату и время.

> **Предостережение.** По трагическому стечению обстоятельств, все три функции возвращают количество миллисекунд с начала отсчета, а *не* объекты `Date`.
> Чтобы сконструировать дату из компонентов UTC, нужно поступить так:
>
> ```
> const deadline = new Date(Date.UTC(2020, 0 /* January */, 31))
> ```

В классе `Date` имеются методы, копирующие акцессоры чтения и записи в Java, например `getHours` и `setHours`, а не акцессоры `get` и `set` в духе JavaScript .

Для получения компонентов объекта `Date` служат методы `getUTCFullYear`, `getUTCMonth` (от 0 до 11), `getUTCDate` (от 1 до 31), `getUTCHours` (от 0 до 23), `getUTCMinutes`, `getUTCSeconds`, `getUTCMilliseconds`.

Методы без слова `UTC` (т. е. `getFullYear`, `getMonth`, `getDate` и т. д.) дают ту же информацию в местном времени. Они нужны разве что для показа местного времени пользователю. Но для этой цели следует использовать какой-нибудь метод форматирования дат из раздела 5.9.

Метод `getUTCDay` возвращает день недели от 0 (воскресенье) до 6 (суббота):

```
const epoch = new Date('1970-01-01T00:00:00.000Z')
epoch.getUTCDay() // 4 (четверг)
epoch.getDay() // 3, 4 или 5 в зависимости от того, где и когда вызван метод
```

> **Примечание.** Устаревший метод `getYear` возвращает двузначный год. Очевидно, в 1995 году, когда был создан JavaScript, никто не мог предположить, что двузначный год может стать проблемой.

JavaScript повторил ошибку Java, сделав объекты `Date` изменяемыми, и даже усугубил ее, предоставив акцессоры записи для каждой компоненты времени (см. упражнение 16). При записи акцессоры молчаливо исправляют ошибки, выбирая следующую корректную дату:

```
const appointment = new Date('2020-05-31T00:00:00.000Z')
appointment.setUTCMonth(5 /* июнь */) // теперь дата встречи 1 июля
```

5.9. Форматирование дат

Методы `toString`, `toDateString`, `toTimeString` и `toUTCString` возвращают строки в «понятном человеку» формате, который на самом деле не особенно понятен:

```
'Sun Jul 20 1969 21:17:40 GMT+0100 (Mitteleuropäische Sommerzeit)'
'Sun Jul 20 1969'
'21:17:40 GMT+0100 (Mitteleuropäische Sommerzeit)'
'Sun, 20 Jul 1969 20:17:40 GMT'
```

Обратите внимание, что часовой пояс (в отличие от дня недели и названия месяца) выводится в пользовательской локали.

Чтобы дата и время были действительно понятны пользователю, используйте методы `toLocaleString`, `toLocaleDateString` или `toLocaleTimeString`, которые форматируют дату и время, только дату или только время соответственно. При этом применяются правила из текущей пользовательской локали или из той локали, которую вы зададите:

```
moonlanding.toLocaleDateString() // '20.7.1969', если локаль German
moonlanding.toLocaleDateString('en-US') // '7/20/1969'
```

Формат, подразумеваемый по умолчанию, довольно короткий, но его можно изменить, указав параметры форматирования:

```
moonlanding.toLocaleDateString(
  'en-US', { year: 'numeric', month: 'long', day: 'numeric' })
  // 'July 20, 1969'
```

Понятие локали объясняется в главе 8, там же эти параметры описаны подробно.

Для получения машиночитаемых дат вызовите просто метод `toISOString`, который возвращает строку в формате ISO 8601:

```
moonlanding.toISOString() // '1969-07-20T20:17:40.000Z'
```

Упражнения

1. Значения 0 и –0 в стандарте IEEE 754 различаются. Предложите по крайней мере две реализации функции `plusMinusZero(x)`, которая возвращает

`+1`, если `x` равно `0`, `-1`, если **x** равно `-0`, и `0` в противном случае. Указание: `Object.is`, `1/-0`.

2. В стандарте IEEE 754 определено три вида значений с плавающей точкой «двойной точности»:
   - «нормализованные» значения вида $\pm 1.m \times 2^e$, где m содержит 52 бита, а e принадлежит диапазону от −1022 до 1023;
   - ± 0 и «денормализованные» значения, близкие к нулю, вида $\pm 0.m \times 2^{-1022}$, где m содержит 52 бита;
   - специальные значения $\pm\infty$, NaN.

   Напишите функцию, которая возвращает строку `'normalized'`, `'denormalized'` или `'special'` для заданного числа с плавающей точкой.
3. Предположим, что число `x`, записанное в экспоненциальном формате, имеет показатель степени `e`. Сформулируйте условие, зависящее от `e` и `p`, при котором вызов `x.toPrecision(p)` показывает результат в формате с фиксированной точкой.
4. Напишите функцию, которая форматирует числовое значение согласно спецификации в стиле `printf`. Например, вызов `format(42, "%04x")` должен напечатать `002A`.
5. Напишите функцию, которая возвращает показатель степени числа с плавающей точкой, т. е. значение, которое будет напечатано после `e` в экспоненциальном формате. Воспользуйтесь двоичным поиском, не обращайтесь к методам из классов `Math` или `Number`.
6. Объясните, почему значения `Number.MAX_VALUE`, `Number.MIN_VALUE` и `Number.EPSILON` (см. раздел 5.4) именно таковы.
7. Напишите функцию, которая вычисляет наименьшее представимое число с плавающей точкой, следующее после заданного целого `n`. Указание: каково наименьшее представимое число после 1? А после 2? А после 3? А после 4? Можете обратиться к какой-нибудь статье, описывающей представление чисел с плавающей точкой в стандарте IEEE. Дополнительные баллы, если сможете получить результат для произвольного числа.
8. Создайте большое целое, в котором цифра 3 повторяется тысячу раз, не прибегая ни к циклам, ни к рекурсии, в одной строчке кода длиной не более 80 знаков.
9. Напишите функцию, которая преобразует объект `Date` в объект со свойствами `year`, `month`, `day`, `weekday`, `hours`, `minutes`, `seconds`, `millis`.
10. Напишите функцию, которая определяет, на сколько часов местное время пользователя отличается от UTC.
11. Напишите функцию, которая определяет, является ли год високосным. Предложите две разные реализации.
12. Напишите функцию, которая возвращает день недели заданной даты, не обращаясь к методам `Date.getUTCDay` и `getDay`. Указание: начало отсчета приходится на четверг.
13. Напишите функцию, которая, получив месяц и год (по умолчанию равные текущим месяцу и году), печатает календарь вида:

```
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

14. Напишите функцию, принимающую два параметра `Date`, которая возвращает количество дней между соответствующими датами, причем дробная часть должна описывать дробную часть дня.
15. Напишите функцию, принимающую два параметра `Date`, которая возвращает количество лет между соответствующими датами. Эта задача труднее предыдущей, потому что количество дней в году переменно.
16. Предположим, что задан следующий крайний срок, а вы хотите отодвинуть его на 1 февраля:

    ```
    const deadline = new Date(Date.UTC(2020, 0 /* январь */, 31))
    ```

    Каким будет результат следующих вызовов?

    ```
    deadline.setUTCMonth(1 /* февраль */)
    deadline.setUTCDate(1)
    ```

    Быть может, нужно всегда вызывать `setUTCDate` до `setUTCMonth`? Приведите пример, показывающий, что эта идея не работает.
17. Поэкспериментируйте со строками, которые принимает метод `Date.parse(dateString)` или конструктор `new Date(dateString)` в своей любимой исполняющей среде JavaScript. Вот несколько примеров:

    ```
    строка, возвращенная Date()
    '3/14/2020'
    'March 14, 2020'
    '14 March 2020'
    '2020-03-14'
    '2020-03-14 '
    ```

    Как это ни ужасно, последние две строки дают *разные* даты в Node.js версии 13.11.0.

Глава 6

Строки и регулярные выражения

В этой главе мы расскажем о методах обработки строк, имеющихся в стандартной библиотеке. Затем обратимся к регулярным выражениям, которые позволяют находить строки, соответствующие образцам. После введения в синтаксис регулярных выражений (с причудами, свойственными JavaScript) будет показано, как использовать этот API для поиска и замены соответствий.

6.1. Преобразование между строками и последовательностями кодовых точек

Строка – это последовательность кодовых точек Юникода. Каждая кодовая точка – это целое число от 0 до `0x10FFFF`. Функция `fromCodePoint` класса `String` собирает строку из кодовых точек, переданных в качестве аргументов:

```
let str = String.fromCodePoint(0x48, 0x69, 0x20, 0x1F310, 0x21) // 'Hi 🌐!'
```

Если кодовые точки помещены в массив, воспользуемся оператором расширения:

```
let codePoints = [0x48, 0x69, 0x20, 0x1F310, 0x21]
str = String.fromCodePoint(...codePoints)
```

И наоборот, можно преобразовать строку в массив кодовых точек:

```
let characters = [...str] // [ 'H', 'i', ' ', '🌐', '!' ]
```

Результатом будет массив строк, каждая из которых содержит одну кодовую точку. Можно получить кодовые точки в форме целых чисел:

```
codePoints = [...str].map(c => c.codePointAt(0))
```

> **Предостережение.** В JavaScript строки хранятся как последовательности кодовых единиц в кодировке UTF-16. Смещение в вызове типа `'Hi 🌐'.codePointAt(i)` относится к кодировке UTF-16. В этом примере допустимы смещения 0, 1, 2, 3 и 5. Если смещение попадет в середину пары кодовых единиц, составляющих одну кодовую точку, то будет возвращена некорректная кодовая точка.

Если вы хотите обойти кодовые точки строки, не помещая их в массив, воспользуйтесь таким циклом:

```
for (let i = 0; i < str.length; i++) {
  let cp = str.codePointAt(i)
  if (cp > 0xFFFF) i++
  ... // обработать кодовую точку cp
}
```

6.2. Подстроки

Метод `indexOf` возвращает индекс первого вхождения подстроки:

```
let index = 'Hello yellow'.indexOf('el') // 1
```

Метод `lastIndexOf` возвращает индекс последнего вхождения:

```
index = 'Hello yellow'.lastIndexOf('el') // 7
```

Как и все смещения в строках JavaScript, эти значения являются смещениями в кодировке UTF-16:

```
index = 'I💛yellow'.indexOf('el') // 4
```

Смещение равно 4, потому что эмоджи «желтое сердце» 💛 в UTF-16 кодируется двумя кодовыми единицами.

Если подстрока отсутствует, этот метод возвращает `-1`.

Методы `startsWith`, `endsWith` и `includes` возвращают булево значение:

```
let isHttps = url.startsWith('https://')
let isGif = url.endsWith('.gif')
let isQuery = url.includes('?')
```

Метод `substring` извлекает подстроку, получив два смещения в кодовых единицах UTF-16. Подстрока содержит все символы, начиная с первого смещения, вплоть до второго смещения, не включая его самого.

```
let substring = 'I💛yellow'.substring(3, 7) // 'yell'
```

Если опустить второе смещение, то включаются все символы до конца строки:

```
substring = 'I💛yellow'.substring(3) // 'yellow'
```

Метод `slice` похож на `substring`, но отрицательное смещение отсчитывается от конца строки. `-1` – смещение последней кодовой единицы, `-2` – пред-

последней и т. д. Для этого длина строки складывается с отрицательным смещением.

```
'I💛yellow'.slice(-6, -2) // 'yell', то же, что slice(3, 7)
```

Длина строки 'I💛yellow' равна 9 – напомним, что 💛 занимает две кодовые единицы. Смещения –6 и –2 заменяются на 3 и 7.

В обоих методах `substring` и `slice` смещения, большие длины строки, заменяются длиной строки. Отрицательные и равные `NaN` смещения заменяются на 0. (В методе `slice` это делается после прибавления длины строки к отрицательному смещению.)

> **Предостережение.** Если первый аргумент `substring` больше второго, то аргументы переставляются!
>
> ```
> substring = 'I💛yellow'.substring(7, 3) // 'yell', то же, что substring(3, 7)
> ```
>
> С другой стороны, метод `str.slice(start, end)` возвращает пустую строку, если `start ≥ end`.

Лично я предпочитаю метод `slice`. Он более гибкий, ведет себя разумнее, да и имя короче.

Еще один способ разобрать строку на части – метод `split`. Он создает из строки массив подстрок, удаляя переданный ему разделитель.

```
let parts = 'Mary had a little lamb'.split(' ')
  // ['Mary', 'had', 'a', 'little', 'lamb']
```

Количество частей можно ограничить:

```
parts = 'Mary had a little lamb'.split(' ', 4)
  // ['Mary', 'had', 'a', 'little']
```

Разделитель может быть регулярным выражением (см. раздел 6.12).

> **Предостережение.** Вызов `str.split('')` с пустым разделителем разбивает строку на строки, каждая из которых содержит одну 16-разрядную кодовую единицу, что не очень полезно, если строка `str` содержит символы старше `\u{FFFF}`. Используйте вместо этого `[...str]`.

6.3. Прочие методы класса String

В этом разделе мы рассмотрим различные методы класса `String`. Поскольку в JavaScript строки неизменяемые, ни один из строковых методов не модифицирует содержимое строки. Все они возвращают новую строку.

Метод `repeat` возвращает строку, повторенную заданное число раз:

```
const repeated = 'ho '.repeat(3) // 'ho ho ho '
```

Методы `trim`, `trimStart` и `trimEnd` возвращают строки, из которых удалены начальные и конечные, только начальные или только конечные пробельные

символы. К числу пробельных символов относятся пробел, неразрывный пробел \u{00A0}, знак новой строки, знак табуляции и еще 21 символ Юникода со свойством White_Space.

Методы padStart и padEnd делают прямо противоположное – добавляют пробелы в начало или в конец короткой строки, так чтобы она достигла заданной длины:

```
let padded = 'Hello'.padStart(10) // '    Hello', добавлено пять пробелов
```

Можно также задать собственную дополняющую строку:

```
padded = 'Hello'.padStart(10, '=-') // =-=-=Hello
```

> **Предостережение.** Первый параметр – длина дополненной строки в байтах. Если дополняющая строка содержит символы, занимающие два байта, то может получиться некорректная строка:
>
> ```
> padded = 'Hello'.padStart(10, '💛')
>   // Дополнена двумя сердечками и непарной кодовой единицей
> ```

Методы toUpperCase и toLowerCase возвращают строку, преобразованную в верхний или нижний регистр соответственно.

```
let uppercased = 'Straße'.toUpperCase() // 'STRASSE'
```

Как видим, метод toUpperCase знает о том, что в верхнем регистре немецкая буква ß записывается строкой 'SS'.

Заметим, что метод toLowerCase не восстанавливает исходную строку:

```
let lowercased = uppercased.toLowerCase() // 'strasse'
```

> **Примечание.** Строковые операции, в частности преобразование в верхний или нижний регистр, могут зависеть от языковых предпочтений пользователя. В главе 8 описаны методы toLocaleUpperCase, toLocaleLowerCase, localeCompare и normalize, полезные для локализации приложений.

> **Примечание.** В разделе 6.12 описаны методы match, matchAll, search и replace для работы с регулярными выражениями.

Метод concat конкатенирует строку с любым количеством аргументов, которые преобразуются в строки.

```
const n = 7
let concatenated = 'agent'.concat(' ', n) // 'agent 7'
```

Того же результата можно достичь с помощью шаблонных строк или метода join класса Array:

```
concatenated = `agent ${n}`
concatenated = ['agent', ' ', n].join('')
```

В табл. 6.1 перечислены наиболее полезные средства класса String.

Таблица 6.1. Полезные функции и методы класса `String`

Имя	Описание
Функции	
`fromCodePoint(codePoints...)`	Возвращает строку, содержащую заданные кодовые точки
Методы	
`startsWith(s)`, `endsWith(s)`, `includes(s)`	`true`, если строка начинается строкой `s`, заканчивается строкой `s` или содержит `s` в качестве подстроки
`indexOf(s, start)`, `lastIndexOf(s, start)`	Индекс соответственно первого или последнего вхождения строки `s`, начиная с позиции `start` (по умолчанию равной 0)
`slice(start, end)`	Подстрока, состоящая из кодовых единиц между индексами `start` (включая) и `end` (не включая). Отрицательные индексы отсчитываются от конца строки. По умолчанию `end` равно длине строке. Этот метод предпочтительнее `substring`
`repeat(n)`	Данная строка, повторенная `n` раз
`trimStart()`, `trimEnd()`, `trim()`	Данная строка, из которой удалены соответственно начальные, конечные или начальные и конечные пробельные символы
`padStart(minLength, padString)`, `padEnd(minLength, padString)`	Данная строка, дополненная в начале или в конце до длины `minLength`. По умолчанию `padString` равна `' '`
`toLowerCase()`, `toUpperCase()`	Данная строка, все символы которой преобразованы соответственно в нижний или верхний регистр
`split(separator, maxParts)`	Массив частей строки, полученный удалением всех вхождений разделителя `separator` (который может быть регулярным выражением). Если параметр `maxParts` опущен, возвращаются все части
`search(target)`	Индекс первого вхождения параметра `target` (который может быть регулярным выражением)
`replace(target, replacement)`	Данная строка, в которой заменено первое вхождение `target`. Если `target` – глобальное регулярное выражение, то заменяются все соответствия ему. О паттернах и функциях замены см. раздел 6.13
`match(regex)`	Возвращает массив соответствий, если `target` – глобальное регулярное выражение, `null`, если соответствий не найдено, и результат сопоставления в противном случае. Результатом сопоставления является массив всех групповых соответствий со свойствами `index` (индекс начала соответствия) и `groups` (объект, отображающий имена групп на соответствия)
`matchAll(regex)`	Итерируемый объект с результатами сопоставлений

Наконец, имеются глобальные функции для приведения компонентов URL, полных URL и более общих URI с такими схемами, как `mailto` или `tel`, к «URL-кодированной» форме. В этой форме допустимы только литеры, признанные «безопасными» во времена, когда создавался интернет. Пусть требуется создать запрос для перевода фразы с одного языка на другой. Тогда можно построить URL следующим образом:

```
const phrase = 'à coté de'
const prefix = 'https://www.linguee.fr/anglais-francais/traduction'
const suffix = '.html'
const url = prefix + encodeURIComponent(phrase) + suffix
```

В результате получим фразу `'%C3%A0%20cot%C3%A9%20de'`, которая является результатом представления литер в кодировке UTF-8 и записи каждого байта своим шестнадцатеричным кодом `%hh`. Остались только «безопасные» литеры из множества

```
A-Z a-z 0-9 ! ' ( ) * . _ ~ -
```

В менее общем случае, когда нужно закодировать URI целиком, пользуйтесь функцией `encodeURI`. Она оставляет неизменными также литеры

```
# $ & + , / : ; = ? @
```

поскольку в составе URI у них имеется специальная семантика.

6.4. Тегированные шаблонные литералы

В главе 1 мы встречались с шаблонными литералами – строками с погруженными в них выражениями:

```
const person = { name: 'Harry', age: 42 }
message = `Next year, ${person.name} will be ${person.age + 1}.`
```

Шаблонный литерал подставляет значения погруженных выражений в строку шаблона. В примере выше погруженные выражения `person.name` и `person.age + 1` вычисляются, преобразуются в строки и вставляются между окружающими фрагментами строки. В результате получается строка

```
'Next year, Harry will be 43.'
```

Мы можем настроить поведение шаблонных литералов с помощью теговой функции. Например, добавим теговую функцию `strong`, которая порождает HTML-строку, выделяющую погруженные значения. Вызов

```
strong`Next year, ${person.name} will be ${person.age + 1}.`
```

возвращает HTML-строку

```
'Next year, <strong>Harry</strong> will be <strong>43</strong>.'
```

Теговой функции передаются фрагменты литеральной строки, окружающие погруженные выражения, за которыми следуют значения выражений. В нашем примере имеются фрагменты `'Next year, '`, `' will be '` и `'.'`, а значениями являются `'Harry'` и `43`. Теговая функция объединяет эти части. Возвращенное значение преобразуется в строку, если еще не является таковой.

В качестве примера приведем реализацию теговой функции `strong`:

```
const strong = (fragments, ...values) => {
  let result = fragments[0]
  for (let i = 0; i < values.length; i++)
```

```
    result += `<strong>${values[i]}</strong>${fragments[i + 1]}`
  return result
}
```

В процессе обработки шаблонной строки

```
strong`Next year, ${person.name} will be ${person.age + 1}.`
```

функция `strong` вызывается следующим образом:

```
strong(['Next year, ', ' will be ', '.'], 'Harry', 43)
```

Заметим, что все фрагменты строки помещены в массив, а значения выражений передаются в отдельных аргументах. Функция `strong` пользуется оператором расширения, чтобы собрать их во второй массив.

Также отметим, что фрагментов всегда на один больше, чем значений выражений.

Этот механизм бесконечно гибкий. Его можно использовать в HTML-шаблонах, для форматирования чисел, интернационализации и т. д.

6.5. Простые шаблонные литералы

Если предпослать шаблонному литералу `String.raw`, то знаки обратной косой черты перестают быть управляющими символами:

```
path = String.raw`c:\users\nate`
```

Здесь `\u` – не управляющий код последовательности Юникода, а `\n` не преобразуется в знак новой строки.

> **Предостережение.** Даже в простом режиме произвольные строки нельзя заключать в обратные кавычки. Все равно необходимо экранировать все литеры `, $ перед { и \ перед ` и {.

Впрочем, это не полное объяснение того, как работает `String.raw`. У теговых функций есть доступ к «простой» форме фрагментов шаблонной строки, в которой комбинации с обратной косой чертой, например `\u` и `\n`, теряют специальный смысл.

Допустим, что мы хотим обрабатывать строки, содержащие греческие буквы. Мы следуем соглашениям, принятым в языке разметки математических формул LATEX. В этом языке символы начинаются знаком обратной косой черты. Поэтому простые строки были бы удобны – пользователи предпочитают писать `\nu` и `\upsilon`, а не `\\nu` и `\\upsilon`. Приведем пример строки, которую мы хотели бы обработать:

```
greek`\nu=${factor}\upsilon`
```

Как и для любой тегированной шаблонной строки, необходимо определить функцию:

```
const greek = (fragments, ...values) => {
  const substitutions = { alpha: 'α', ..., nu: 'ν', ... }
  const substitute = str => str.replace(/\\[a-z]+/g,
    match => substitutions[match.slice(1)])
  let result = substitute(fragments.raw[0])
  for (let i = 0; i < values.length; i++)
    result += values[i] + substitute(fragments.raw[i + 1])
  return result
}
```

Для доступа к простым фрагментам строки служит свойство `raw` первого параметра теговой функции. Значением `fragments.raw` является массив фрагментов строки с необработанными знаками обратной косой черты.

В показанном выше тегированном шаблонном литерале `fragments.raw` – массив из двух строк: `\nu=` и `\upsilon`.

Отметим следующие моменты:

- `\n` в `\nu` не преобразуется в знак новой строки;
- `\u` в `\upsilon` не интерпретируется как управляющий символ Юникода. Да это и синтаксически было бы некорректно. Поэтому `fragments[1]` невозможно разобрать, и этому элементу присваивается значение `undefined`;
- `${factor}` – погруженное выражение. Его значение передается теговой функции.

В функции `greek` используется замена регулярного выражения, подробно рассматриваемая в разделе 6.13. Вместо идентификаторов, начинающихся знаком обратной косой черты, подставляются соответствующие символы, например ν вместо `\nu`.

6.6. Регулярные выражения

Регулярные выражения задают образцы для сопоставления со строками. Они используются, когда нужно найти строки, отвечающие определенному образцу. Пусть, например, требуется найти гиперссылки в HTML-файле. Мы ищем строки вида `<a href="...">`. Но секундочку – ведь могут присутствовать дополнительные пробелы, а URL может быть заключен в одиночные кавычки. Регулярные выражения предлагают синтаксические конструкции для определения допустимых последовательностей символов.

В регулярном выражении символ обозначает сам себя, если не является одним из специальных символов

```
. * + ? { | ( ) [ \ ^ $
```

Например, регулярное выражение `href` сопоставляется только со строкой `href`.

Символ `.` сопоставляется с любым символом, но только одним. Например, `.r.f` сопоставляется с `href` и с `prof`.

Символ * означает, что предшествующая ему конструкция может быть повторена 0 или более раз, а символ + – что повторений может быть 1 или более. Суффикс ? означает, что предшествующая конструкция факультативная (может встречаться 0 или 1 раз). Например, `be+s?` сопоставляется с `be`, `bee` и `bees`. Другое число повторений можно задать с помощью фигурных скобок { } (см. табл. 6.2).

Символ | означает альтернативу: `.(oo+|ee+)f` сопоставляется с `beef` или `woof`. Обратите внимание на круглые скобки – без них `.oo+|ee+f` было бы альтернативой между `.oo+` и `ee+f`. Скобки используются также для группировки – см. раздел 6.11 «Группы».

Классом символов называется множество символов, заключенное в квадратные скобки, например `[Jj]`, `[0-9]`, `[A-Za-z]` или `[^0-9]`. Внутри класса символов знак - обозначает диапазон (все символы, значения Юникода которых находятся между нижней и верхней границами). Однако если - является первым или последним символом в классе, то он обозначает сам себя. Знак ^ в начале класса символов означает дополнение – все символы, кроме указанных. Например, `[^0-9]` означает все символы, кроме десятичных цифр.

Существует шесть *предопределенных классов символов*: `\d` (цифры), `\s` (пробельные символы), `\w` (символы, являющиеся частью слова), а также их дополнения: `\D` (не цифры), `\S` (не пробельные символы) и `\W` (символы, не являющиеся частью слова).

Символы ^ и $ сопоставляются с началом и концом входных данных. Например, `^[0-9]+$` сопоставляется со строкой, целиком состоящей из цифр.

Следите за тем, где находится символ ^. Если это первый символ внутри квадратных скобок, то он обозначает дополнение: `[^0-9]+$` сопоставляется со строкой не цифр в конце входной строки.

> **Примечание.** Я долго не мог запомнить, что ^ сопоставляется с началом, а $ – с концом строки. Я и сейчас думаю, что $ должен означать start (начало), да и на американской клавиатуре $ находится левее ^. Но на самом деле все в точности наоборот – возможно, потому что в древнем редакторе QED знак $ обозначал последнюю строку.

В табл. 6.2 приведена синтаксическая сводка регулярных выражений в JavaScript. Если вам нужно включить в регулярное выражение один из символов `. * + ? { | ( ) [ \ ^ $` в своем исконном смысле, то поставьте перед ним знак обратной косой черты. Внутри класса символов нужно экранировать только [и \, при условии что вы ничего не напутали с позициями символов] - ^. Например, `[]^-]` – класс, содержащий все три этих символа.

Таблица 6.2. Синтаксис регулярных выражений

Выражение	Описание	Пример
Символы		
Символ, отличный от `. * + ? { | ( ) [ \ ^ $`	Сопоставляется только с самим собой	`J`
`.`	Сопоставляется с любым символом, кроме `\n`, или вообще с любым символом, если поднят флаг `dotAll`	

Таблица 6.2 *(продолжение)*

Выражение	Описание	Пример
\u{*hhhh*}, \u{*hhhhh*}	Кодовая точка Юникода с указанным шестнадцатеричным значением (требуется флаг unicode)	\u{1F310}
\u*hhhh*, \x*hh*	Кодовая точка UTF-16 с указанным шестнадцатеричным значением	\xA0
\f, \n, \r, \t, \v	Подача страницы (\x0C), новая строка (\x0A), возврат каретки (\x0D), табуляция (\x09), вертикальная табуляция (\x0B)	\n
\c*L*, где *L* принадлежит [A-Za-z]	Управляющий символ, соответствующий символу *L*	\cH – это **Ctrl**+**H**, или забой (\x08)
c, где *c* не принадлежит [0-BDPSWbcdfknprstv]	Символ *c*	\\
Классы символов		
[C_1C_2...], где C_i – символы, диапазоны вида *c-d* или классы символов	Любой из символов, представленных C_1, C_2, ...	[0-9+-]
[^...]	Дополнение класса символов	[^\d\s]
\p{*БулевоСвойство*} \p{*Свойство=Значение*} \P{...}	Свойство Юникода (см. раздел 6.9); его дополнение (требуется флаг unicode)	\p{L} – буквы Юникода
\d, \D	Цифры [0-9]; дополнение	\d+ – последовательность цифр
\w, \W	Символ, являющийся частью слова [a-zA-Z0-9_]; дополнение	
\s, \S	Пробельный символ из класса [\t\n\v\f\r\xA0] или любой из 18 пробельных символов Юникода; то же, что \p{White_Space}	\s*,\s* – запятая, окруженная необязательными пробельными символами
Последовательности и альтернативы		
XY	Любая строка из *X*, за которой следует любая строка из *Y*	[1-9][0-9]* – положительное число без начального нуля
X|*Y*	Любая строка из *X* или *Y*	http|ftp
Группировка		
(*X*)	Запоминает сопоставление с *X* в группе (см. раздел 6.11)	'([^']*)' запоминает заключенный в кавычки текст
n	Сопоставляется с *n*-й группой	(['"]).*\1 сопоставляется с 'Fred' или "Fred", но не с "Fred'
(?<*name*>*X*)	Запоминает сопоставление с *X* под указанным именем	'(?<qty>[0-9]+)' запоминает сопоставление под именем qty
\k<*name*>	Группа с указанным именем	\k<qty> сопоставляется с группой с именем qty

Таблица 6.2 *(окончание)*

Выражение	Описание	Пример
(?:*X*)	Использование скобок без запоминания *X*	В выражении (?:http|ftp)://(.*) сопоставленная строка после :// возвращается в виде \1
Прочие конструкции (?...)	См. раздел 6.14	
Кванторы		
X?	Необязательное вхождение *X*	\+? – необязательный вопросительный знак
*X**, *X*+	0 или более *X*; 1 или более *X*	[1-9][0-9]+ – целое число ⩾ 10
X{*n*}, *X*{*n*,}, *X*{*m*,*n*}	*n* раз *X*; по меньшей мере *n* раз *X*; от *m* до *n* раз *X*	[0-9]{4,6} – от четырех до шести цифр
*X**? или *X*+?	Нежадный квантор, пробующий самые короткие соответствия, перед тем как переходить к более длинным	.*(<.+?>).* запоминает самую короткую последовательность литер, заключенную в угловые скобки
Сопоставление с границей		
^ $	Начало и конец входной строки (или строчки, состоящей из нескольких строк, если поднят флаг multiline)	^JavaScript$ сопоставляется с точной строкой JavaScript
\b, \B	Границы слова, не граница слова	\bJava\B сопоставляется с JavaScript, но не с Java code

6.7. Литеральные регулярные выражения

Литеральное регулярное выражение ограничено знаками косой черты:

```
const timeRegex = /^([1-9]|1[0-2]):[0-9]{2} [ap]m$/
```

Литеральные регулярные выражения являются экземплярами класса RegExp.

Оператор typeof, примененный к регулярному выражению, возвращает 'object'.

Внутри литерального регулярного выражения следует использовать знаки обратной косой черты для экранирования символов, имеющих специальный смысл в регулярных выражениях, например . и +:

```
const fractionalNumberRegex = /[0-9]+\.[0-9]*/
```

Здесь экранированная . означает буквально точку.

В литеральном регулярном выражении следует экранировать также знак косой черты, чтобы он не интерпретировался как конец литерала.

Для преобразования строки, содержащей регулярное выражение, в объект `RegExp` служит функция `RegExp`, с `new` или без `new`:

```
const fractionalNumberRegex = new RegExp('[0-9]+\\.[0-9]*')
```

Отметим, что знак обратной косой черты в строке следует экранировать.

6.8. Флаги

Флаг модифицирует поведение регулярного выражения. Примером может служить флаг `i`, или `ignoreCase`. Регулярное выражение

```
/[A-Z]+\.com/i
```

сопоставляется со строкой `Horstmann.COM`.

Можно также задать флаг в конструкторе:

```
const regex = new RegExp(/[A-Z]+\.com/, 'i')
```

Чтобы узнать, какие флаги подняты для данного объекта `RegExp`, можно воспользоваться свойством `flags`, возвращающим всю строку флагов. Существуют также булевы свойства для каждого флага в отдельности:

```
regex.flags // 'i'
regex.ignoreCase // true
```

JavaScript поддерживает шесть флагов, показанных в табл. 6.3.

Таблица 6.3. Флаги регулярных выражений

Однобуквенный	Имя свойства	Описание
`i`	`ignoreCase`	Сопоставление с учетом регистра
`m`	`multiline`	`^`, `$` сопоставляется соответственно с началом и концом строчки, а не строки
`s`	`dotAll`	`.` сопоставляется в том числе со знаком новой строки
`u`	`unicode`	Сопоставление с символами Юникода, а не с кодовыми единицами (см. раздел 6.9)
`g`	`global`	Поиск всех соответствий (см. раздел 6.10)
`y`	`sticky`	Соответствие должно начинаться с *regex*.`lastIndex` (см. раздел 6.10)

Флаг `m`, или `multiline`, изменяет поведение якорей начала и конца `^` и `$`. По умолчанию они сопоставляются с началом и концом всей строки. В режиме же `multiline` сопоставление происходит с началом и концом строчки (т. е. части строки, ограниченной слева началом строки или знаком новой строки, а справа – знаком новой строки или концом строки)[1]. Например:

[1] Катастрофическая неоднозначность русского слова «строка» преследует переводчиков технической литературы с английского. Особенно радостно встретить фразу «table row of multiline strings». Хочется надеяться, что читатель по контексту понимает, о какой «строке» идет речь. – *Прим. перев.*

```
/^[0-9]+/m
```

сопоставляется с цифрами в начале строчки.

Если поднят флаг `s` или `dotAll`, то образец `.` сопоставляется со знаками новой строки. Если же флаг сброшен, то `.` сопоставляется с любым символом, кроме знака новой строки.

Остальные три флага объясняются в последующих разделах.

Можно задать сразу несколько флагов. Следующее регулярное выражение сопоставляется со строчными или заглавными буквами в начале каждой строчки:

```
/^[A-Z]/im
```

6.9. Регулярные выражения и Юникод

В силу исторических причин регулярные выражения работают с кодовыми единицами UTF-16, а не с символами Юникода. Например, образец `.` сопоставляется с одной кодовой единицей UTF-16. Следовательно, строка

```
'Hello 🌐'
```

не соответствует регулярному выражению

```
/Hello .$/
```

Символ 🌐 кодируется двумя кодовыми точками. Чтобы исправить ситуацию, задайте флаг `u`, или `unicode`:

```
/Hello .$/u
```

Если флаг `u` поднят, то образец `.` сопоставляется с одним символом Юникода, как бы он ни кодировался в UTF-16.

Если вы хотите хранить исходные файлы в кодировке ASCII, то можете погрузить кодовые точки Юникода в регулярные выражения, воспользовавшись синтаксисом `\u{ }`:

```
/[A-Za-z]+ \u{1F310}/u
```

> **Предостережение.** Без флага `u` выражение `/\u{1F310}/` сопоставляется со строкой `'u{1F310}'`.

При работе с текстом на языках, отличных от английского, следует избегать образцов типа `[A-Za-z]` для обозначения букв. Они не сопоставляются с буквами других языков. Вместо этого пользуйтесь конструкцией `\p{`*Свойство*`}`, где *Свойство* – имя булева свойства Юникода. Например, `\p{L}` обозначает букву Юникода. Регулярное выражение

```
/Hello, \p{L}+!/u
```

сопоставляется с

```
'Hello, värld!'
```

и с

```
'Hello, 世界!'
```

В табл. 6.4 перечислены имена наиболее употребительных свойств Юникода.

Для свойств Юникода с небулевыми значениями используйте конструкцию `\p{`*Свойство*`=`*Значение*`}`. Например, регулярное выражение

```
/p{Script=Han}+/u
```

сопоставляется с любой последовательностью китайских литер.

Заглавная буква `\P` обозначает дополнение: `\P{L}` сопоставляется с любым символом, не являющимся буквой.

Таблица 6.4. Наиболее употребительные булевы свойства Юникода

Имя	Описание
`L`	Буква
`Lu`	Заглавная буква
`Ll`	Строчная буква
`Nd`	Десятичное число
`P`	Знак препинания
`S`	Символ
`White_Space`	Пробельный символ, то же, что `\s`
`Emoji`	Значки эмоджи, модификаторы или компоненты

6.10. Методы класса RegExp

Метод `test` возвращает `true`, если строка *содержит* подстроку, соответствующую данному регулярному выражению:

```
/[0-9]+/.test('agent 007') // true
```

Чтобы проверить, соответствует ли регулярному выражению вся строка целиком, следует использовать якоря начала и конца:

```
/^[0-9]+$/.test('agent 007') // false
```

Метод `exec` возвращает массив, содержащий первое сопоставленное подвыражение, или `null`, если соответствий нет. Например,

```
/[0-9]+/.exec('agents 007 and 008')
```

возвращает массив, содержащий строку `'007'`. (В следующем разделе мы увидим, что массив может содержать также групповые соответствия.)

Кроме того, массив, возвращенный методом `exec`, обладает двумя свойствами:

- index – индекс начала подвыражения;
- input – аргумент, переданный exec.

Иными словами, предыдущее обращение к exec на самом деле вернуло такой массив:

```
['007', index: 7, input: 'agents 007 and 008']
```

Для поиска всех сопоставленных подвыражений задайте флаг g, или global:

```
let digits = /[0-9]+/g
```

Теперь каждое обращение к exec возвращает новое соответствие:

```
result = digits.exec('agents 007 and 008') // ['007', index: 7, . . .]
result = digits.exec('agents 007 and 008') // ['008', index: 15, . . .]
result = digits.exec('agents 007 and 008') // null
```

Чтобы этот код мог работать, в объекте RegExp имеется свойство lastIndex, значением которого является индекс следующего за соответствием символа после каждого успешного обращения к exec. При следующем обращении к exec поиск начнется с позиции lastIndex. Свойству lastIndex присваивается 0 при конструировании регулярного выражения или отсутствии соответствия.

Можно также самостоятельно установить свойство lastIndex, чтобы пропустить часть строки.

Если поднят флаг y, или sticky, то соответствие должно начинаться *точно* в позиции lastIndex:

```
digits = /[0-9]+/y
digits.lastIndex = 5
result = digits.exec('agents 007 and 008') // null
digits.lastIndex = 8
result = digits.exec('agents 007 and 008') // ['07', index: 8, . . .]
```

> **Примечание.** Если нужно просто получить массив всех сопоставленных подстрок, пользуйтесь методом match класса String, а не повторными обращениями к exec (см. раздел 6.12).
>
> ```
> let results = 'agents 007 and 008'.match(/[0-9]+/g) // ['007', '008']
> ```

6.11. Группы

Группы предназначены для выделения компонентов соответствия. Например, ниже показано регулярное выражение для разбора времени, содержащее группы для каждой компоненты:

```
let time = /([1-9]|1[0-2]):([0-5][0-9])([ap]m)/
```

Групповые соответствия помещаются в массив, возвращенный методом exec:

```
let result = time.exec('Lunch at 12:15pm')
// ['12:15pm', '12', '15', 'pm', index: 9, . . .]
```

Как было сказано в предыдущем разделе, `result[0]` – это вся сопоставленная строка. Для `i > 0` `result[i]` содержит соответствие `i`-й группе.

Группы нумеруются в порядке следования их *открывающих* скобок. Это существенно, когда имеются вложенные скобки. Рассмотрим пример. Мы хотим проанализировать строки накладных, имеющие вид

```
Blackwell Toaster   USD29.95
```

Вот необходимое нам регулярное выражение с группами для каждой компоненты:

```
/(\p{L}+(\s+\p{L}+)*)\s+([A-Z]{3})([0-9.]*)/u
```

В этой ситуации группа 1 – `'Blackwell Toaster'`, подстрока, сопоставленная выражению `(\p{L}+(\s+\p{L}+)*)`, начинающемуся первой открывающей скобкой и продолжающемуся до парной закрывающей скобки.

Группа 2 – `' Toaster'`, подстрока, сопоставленная выражению `(\s+\p{L}+)`.

Группы 3 и 4 – соответственно `'USD'` и `'29.95'`.

Группа 2 нас не интересует, она возникла только из-за скобок, необходимых, чтобы описать повторение. Для пущей ясности можно воспользоваться незапоминающей группой, добавив `?:` после открывающей скобки:

```
/(\p{L}+(?:\s+\p{L}+)*)\s+([A-Z]{3})([0-9.]*)/u
```

Теперь `'USD'` и `'29.95'` запоминаются в группах 2 и 3.

> **Примечание.** Если группа повторяется, как в случае `(\s+\p{L}+)*` в примере выше, то запоминается только последнее *соответствие*, а не все.
> Если не было найдено ни одного повторения, то сопоставленной группе присваивается значение `undefined`.

Можно произвести сопоставление с содержимым запомненной группы. Например, рассмотрим регулярное выражение:

```
/(['"]).*\1/
```

Группа `(['"])` запоминает одиночную или двойную кавычку. Образец `\1` сопоставляется с запомненной строкой, так что `"Fred"` и `'Fred'` соответствуют регулярному выражению, а `"Fred'` – нет.

> **Предостережение.** Хотя предполагается, что экранирование восьмеричных цифр в строгом режиме запрещено, некоторые движки JavaScript все еще поддерживают их в регулярных выражениях. Например, `\11` обозначает `\t` – символ в кодовой точке 9.

Но если в регулярном выражении 11 или более запоминающих групп, то `\11` обозначает соответствие 11-й группе.

Нумерованные группы чреваты ошибками. Гораздо лучше запоминать группы по имени:

```
let lineItem = /(?<item>\p{L}+(\s+\p{L}+)*)\s+(?<currency>[A-Z]{3})(?<price>[0-9.]*)/u
```

Если в регулярном выражении имеется одна или несколько именованных групп, то массив, возвращенный методом exec, содержит свойство groups, значением которого является объект с именами групп и найденными соответствиями:

```
let result = lineItem.exec('Blackwell Toaster USD29.95')
let groupMatches = result.groups
  // { item: 'Blackwell Toaster', currency: 'USD', price: '29.95' }
```

Выражение \k<*name*> сопоставляется с группой, запомненной по имени:

```
/(?<quote>['"]).*\k<quote>/
```

Здесь группа с именем «quote» сопоставляется с одиночной или двойной кавычкой в начале строки. При этом строка должна заканчиваться тем же символом, которым начиналась. Например, "Fred" и 'Fred' являются соответствиями, а "Fred' – нет.

Средства класса RegExp сведены в табл. 6.5.

Таблица 6.5. Средства класса RegExp

Имя	Описание
Конструкторы	
new RegExp(regex, flags)	Конструирует регулярное выражение из аргумента regex (который может быть строкой, литеральным регулярным выражением или объектом RegExp) с заданными флагами
Свойства	
flags	Строка всех флагов
ignoreCase, multiline, dotAll, unicode, global, sticky	Булевы свойства, соответствующие флагам
Методы	
test(str)	true, если str содержит подстроку, соответствующую данному регулярному выражению
exec(str)	Результаты текущего сопоставления данного регулярного выражения с частью строки str. Подробности см. в разделе 6.10. Методами match и matchAll класса String пользоваться проще, чем exec

6.12. Методы класса String для работы с регулярными выражениями

В разделе 6.10 мы видели, что рабочей лошадкой для получения информации о соответствии является метод exec класса RegExp. Однако его API не может похвастаться элегантностью. В классе String имеется несколько методов для работы с регулярными выражениями, которые в типичных ситуациях позволяют получить результат проще.

Если в регулярном выражении не установлен флаг global, то вызов str.match(regex) возвращает те же результаты, что regex.exec(str):

```
'agents 007 and 008'.match(/[0-9]+/) // ['007', index: 7, ...]
```

Если же флаг global установлен, то match просто возвращает массив соответствий, и часто это именно то, что нужно:

```
'agents 007 and 008'.match(/[0-9]+/g) // ['007', '008']
```

Если соответствий не найдено, то метод String.match возвращает null.

> **Примечание.** RegExp.exec и String.match – единственные методы в стандартной библиотеке ECMAScript, которые возвращают null в качестве индикатора отсутствия результата.

Если вы хотите получить все результаты глобального поиска соответствий, не вызывая повторно exec, то вам понравится метод matchAll класса String, который в настоящее время находится на третьей стадии рассмотрения. Он возвращает итерируемый объект, содержащий результаты сопоставления. Пусть требуется найти все соответствия регулярному выражению

```
let time = /([1-9]|1[0-2]):([0-5][0-9])([ap]m)/g
```

Цикл

```
for (const [, hours, minutes, period] of input.matchAll(time)) {
  ...
}
```

обходит все результаты сопоставления, применяя деструктуризацию для присваивания значений групп переменным hours, minutes и period. Начальная запятая поставлена для того, чтобы игнорировать все сопоставленное выражение целиком.

Метод matchAll отдает соответствия лениво. Это эффективно, когда соответствий много, а нужны лишь несколько первых.

Метод search возвращает индекс первого соответствия или -1, если соответствий не найдено:

```
let index = 'agents 007 and 008'.search(/[0-9]+/) // возвращает индекс 7
```

Метод replace заменяет первое соответствие регулярному выражению строкой замены. Для замены всех соответствий нужно поднять флаг global:

```
let replacement = 'agents 007 and 008'.replace(/[0-9]/g, '?')
  // 'agents ??? and ???'
```

> **Примечание.** Метод split может принимать регулярное выражение в качестве аргумента. Например,
>
> ```
> str.split(/\s*,\s*/)
> ```
>
> разбивает str по запятым, возможно, окруженным пробелами.

6.13. Еще о методе replace

В этом разделе мы ближе познакомимся с методом `replace` класса `String`. Строка замены может содержать образцы, начинающиеся знаком $, которые обрабатываются, как показано в табл. 6.6.

Таблица 6.6. Образцы в строке замены

Образец	Описание
$`, $'	Соответственно часть до и после сопоставленной строки
$&	Сопоставленная строка
$*n*	*n*-я группа
$<*name*>	Группа с именем *name*
$$	Знак доллара

В следующем примере каждая гласная буква повторяется три раза:

```
'hello'.replace(/[aeiou]/g, '$&$&$&') // 'heeellooo'
```

Наиболее полезны образцы групп. В следующем примере группы используются, чтобы выделить в каждой строчке имя и фамилию человека, а затем переставить их местами:

```
let names = 'Harry Smith\nSally Lin'
let flipped = names.replace(
  /^([A-Z][a-z]+) ([A-Z][a-z]+)/gm, "$2, $1")
  // 'Smith, Harry\nLin, Sally'
```

Если число после знака $ больше количества групп в регулярном выражении, то образец копируется буквально:

```
let replacement = 'Blackwell Toaster $29.95'.replace('\$29', '$19')
  // 'Blackwell Toaster $19.95' - нет группы с номером 19
```

Можно также использовать именованные группы:

```
flipped = names.replace(/^(?<first>[A-Z][a-z]+) (?<last>[A-Z][a-z]+)$/gm,
  "$<last>, $<first>")
```

Если нужна более сложная замена, то можно вместо строки замены указать функцию. Функции передаются следующие аргументы:

- строка, сопоставленная регулярному выражению;
- соответствия всем группам;
- смещение соответствия;
- строка целиком.

В примере ниже мы просто обрабатываем соответствия группам:

```
flipped = names.replace(/^([A-Z][a-z]+) ([A-Z][a-z]+)/gm,
  (match, first, last) => `${last}, ${first[0]}.`)
  // 'Smith, H.\nLin, S.'
```

Примечание. Метод replace работает также со строками, при этом заменяется первое совпадение с самой строкой:

```
let replacement = 'Blackwell Toaster $29.95'.replace('$', 'USD')
  // Заменяет $ на USD
```

Обратите внимание, что $ не интерпретируется как якорь конца.

Предостережение. Когда методу search передается строка, она преобразуется в регулярное выражение:

```
let index = 'Blackwell Toaster $29.95'.search('$')
  // возвращает 24, индекс конца строка, а не знака $
```

Для поиска простой подстроки пользуйтесь методом indexOf.

6.14. Экзотические возможности

В последнем разделе этой главы мы рассмотрим несколько продвинутых возможностей регулярных выражений, которые используются нечасто.

Операторы повторения + и * «жадные», т. е. ищут самое длинное возможное соответствие. Обычно это именно то, что нужно. Мы хотим, чтобы регулярное выражение /[0-9]+/ сопоставлялось с самой длинной строкой цифр, а не с одной цифрой.

Однако рассмотрим следующий пример:

```
'"Hi" and "Bye"'.match(/".*"/g)
```

Результатом будет

```
'"Hi" and "Bye"'
```

поскольку .* жадно сопоставляется со всеми литерами вплоть до последней кавычки ". Это плохо, если мы хотим искать заключенные в кавычки подстроки.

Проблему можно решить, если потребовать, чтобы среди повторяющихся литер не было кавычек:

```
'"Hi" and "Bye"'.match(/"[^"]*"/g)
```

Или же можно указать, что сопоставление должно быть *нежадным*, воспользовавшись оператором *?:

```
'"Hi" and "Bye"'.match(/".*?"/g)
```

В обоих случаях все закавыченные строки сопоставляются по отдельности и получается

```
['"Hi"', '"Bye"']
```

Существует также нежадная версия +?, которой достаточно по меньшей мере одного повторения.

Оператор *заглядывания вперед* `p(?=q)` сопоставляется с *p* при условии, что далее следует *q*, но не включает *q* в найденное соответствие. В примере ниже мы ищем час, которому должно предшествовать двоеточие:

```
let hours = '10:30 - 12:00'.match(/[0-9]+(?=:)/g) // ['10, 12']
```

Инвертированный оператор заглядывания вперед `p(?!q)` сопоставляется с *p*, если далее *не следует q*.

```
let minutes = '10:30 - 12:00'.match(/[0-9][0-9](?!:)/g) // ['10, 12']
```

Существует также оператор *оглядывания назад* `(?<=p)q`, который сопоставляется с *q* при условии, что перед ним находится *p*.

```
minutes = '10:30 - 12:00'.match(/(?<=[0-9]+:)[0-9]+/g) // ['30', '00']
```

Заметим, что аргумент внутри `(?<=[0-9]+:)` сам является регулярным выражением. Наконец, существует инвертированный оператор оглядывания назад `(?<!p)q`, который сопоставляется с *q* при условии, что ему не предшествует *p*.

```
hours = '10:30 - 12:00'.match(/(?<![0-9:])[0-9]+/g)
```

Наверное, такие регулярные выражения стали причиной бессмертного высказывания Джеми Завински: «Некоторые, столкнувшись с проблемой, думают: “А, ничего страшного, воспользуюсь регулярным выражением”. И теперь у них на руках две проблемы».

Упражнения

1. Напишите функцию, которая, получив строку, возвращает экранированную строку, заключенную в одиночные кавычки `'`. Все символы Юникода, отличные от ASCII, представьте в виде `\u{...}`. Специальные символы экранируйте в виде `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\'`, `\\`.
2. Напишите функцию, которая ограничивает строку заданным числом символов Юникода. Если строка слишком длинная, обрежьте ее и добавьте в конце знак многоточия … (`\u{2026}`). Корректно обрабатывайте символы, кодируемые двумя кодовыми единицами UTF-16.
3. Методы `substring` и `slice` очень терпимо относятся к неправильным аргументам. Сможете ли вы привести пример, когда они возвращают ошибку из-за неправильного аргумента? Попробуйте строки, объекты, массивы, отсутствие аргументов.
4. Напишите функцию, которая принимает строку и возвращает массив всех подстрок. Будьте внимательны к символам, кодируемым двумя кодовыми единицами UTF-16.
5. В более совершенном мире все методы работы со строками принимали бы смещения, равные количеству символов Юникода, а не кодовых

единиц UTF-16. Какие методы класса `String` были бы затронуты таким изменением? Предложите замены для них, например `indexOf(str, sub)` и `slice(str, start, end)`.

6. Напишите теговую шаблонную функцию `printf`, которая форматирует целые числа, числа с плавающей точкой и строки с помощью классических команд форматирования `printf`, которые помещаются после погруженных выражений:

```
const formatted = printf`${item}%-40s | ${quantity}%6d | ${price}%10.2f`
```

7. Напишите теговую шаблонную функцию `spy`, которая отображает простые и обработанные фрагменты строки, а также значения погруженных выражений. Из простых фрагментов строки удалите знаки обратной косой черты, которые были необходимы для экранирования обратных кавычек, знаков долларов и обратной косой черты.
8. Придумайте как можно больше способов создать регулярное выражение, сопоставляемое только с пустой строкой.
9. Действительно ли полезен флаг `m` (`multiline`)? Нельзя ли просто сопоставить с `\n`? Придумайте регулярное выражение, которое может найти все строчки (lines), содержащие только цифры, не пользуясь флагом `multiline`. Как обстоит дело с последней строчкой?
10. Напишите регулярные выражения для адресов электронной почты и URL-адресов.
11. Напишите регулярные выражения для формата телефонных номеров, принятого в США[1], и для международных телефонных номеров.
12. Воспользуйтесь заменой по регулярному выражению для очистки телефонных номеров и номеров банковских карт.
13. Напишите регулярное выражение для выделения текста, заключенного в парные одиночные или двойные кавычки, а также в угловые кавычки «».
14. Напишите регулярное выражение для выделения URL-адресов изображений в HTML-документе.
15. С помощью регулярного выражения выделите все десятичные целые числа (включая отрицательные) в строке и поместите их в массив.
16. Пусть требуется использовать регулярное выражение для сопоставления со всей строкой, а не с ее подстрокой. Казалось бы, нужно лишь добавить в начало `^`, а в конец `$`. Но все не так просто. Прежде чем добавлять эти якоря, регулярное выражение необходимо правильно экранировать. Напишите функцию, которая принимает регулярное выражение и возвращает регулярное выражение с добавленными якорями.
17. Воспользуйтесь методом `replace` класса `String` с аргументом-функцией, чтобы заменить все значения температуры в градусах Фаренгейта (°F) эквивалентными значениями в градусах Цельсия (°C).

---

[1] В США принят такой формат телефонных номеров: +1 (XXX) XXX-XXXX. – *Прим. перев.*

18. Дополните функцию `greek` из раздела 6.5, так чтобы она правильно обрабатывала знаки обратной косой черты и `$`. Также проверяйте, является ли символ, начинающийся знаком обратной косой черты, подстановкой вместо ранее сопоставленной группы. Если нет, включайте его буквально.
19. Обобщите функцию `greek` из предыдущего упражнения, сделав из нее универсальную функцию подстановки, которую можно вызывать следующим образом: `subst(`*словарь*`)`` `*шаблоннаяСтрока*` ``.

Глава 7

Массивы и коллекции

При изучении нового языка программирования всегда возникает вопрос, как хранятся данные. Традиционной структурой для хранения последовательных данных является скромный массив. В этой главе мы узнаем о различных методах массива, предоставляемых JavaScript API. Затем обратимся к типизированным массивам и к буферным массивам – более сложным структурам для эффективного хранения блоков двоичных данных. В отличие от Java или C++, JavaScript не предлагает развитого набора структур данных, но есть простые классы отображения и множества, которые мы обсудим в конце главы.

7.1. Конструирование массива

Вы уже знаете, как сконструировать массив из заданной последовательности элементов – просто написать литерал:

```
const names = ['Peter', 'Paul', 'Mary']
```

А вот как конструируется пустой массив с 10 000 элементов, первоначально равных `undefined`:

```
const bigEmptyArray = []
bigEmptyArray.length = 10000
```

В массивовом литерале могут встречаться расширения любого *итерируемого объекта*. Таковыми являются массивы, строки, множества и отображения, рассматриваемые ниже в этой главе, а также объекты `NodeList` и `HTMLCollection`, входящие в состав DOM API. Вот, например, как формируется массив, содержащий элементы двух итерируемых объектов `a` и `b`:

```
const elements = [...a, ...b]
```

В главе 9 мы увидим, что итерируемые объекты имеют довольно сложную структуру. Функция `Array.from` собирает объекты из более простого *массивоподобного* объекта. Так называется объект, у которого есть целочисленное свойство `'length'`, а также свойства с именами `'0'`, `'1'`, `'2'` и т. д. Конечно, сами массивы являются массивоподобными объектами, но некоторые мето-

ды DOM API возвращают массивоподобные объекты, не являющиеся ни массивами, ни итерируемыми объектами. Чтобы поместить их в массив, нужно вызвать функцию `Array.from(arrayLike)`.

```
const arrayLike = { length: 3 , '0': 'Peter', '1': 'Paul', '2': 'Mary'}
const elements = Array.from(arrayLike)
  // elements - массив ['Peter', 'Paul', 'Mary']
  // Array.isArray(arrayLike) равно false, Array.isArray(elements) равно true
```

Функция `Array.from` принимает необязательный второй аргумент – функцию, которая вызывается для всех значений индекса от 0 до `length - 1`, получая элемент (или `undefined` для отсутствующих элементов) и его индекс. Результаты, возвращенные этой функцией, сохраняются в массиве. Например:

```
const squares = Array.from({ length: 5 }, (element, index) => index * index)
  // [0, 1, 4, 9, 16]
```

Предостережение. Существует конструктор массива из заданных элементов, который можно вызывать с ключевым словом `new` или без него:

```
names = new Array('Peter', 'Paul', 'Mary')
names = Array('Peter', 'Paul', 'Mary')
```

Но тут есть подвох. Вызов `new Array` и `Array` с одним числовым аргументом приводит к совершенно неожиданному результату. Одиночный аргумент обозначает длину массива:

```
numbers = new Array(10000)
```

В результате получается массив длины 10 000, не содержащий ни одного элемента!

Я рекомендую держаться подальше от конструктора `Array` и пользоваться массивовыми литералами:

```
names = ['Peter', 'Paul', 'Mary']
numbers = [10000]
```

Примечание. У фабричной функции `Array.of` нет проблемы, свойственной конструктору `Array`:

```
names = Array.of('Peter', 'Paul', 'Mary')
littleArray = Array.of(10000) // Массив длины 1, то же, что [10000]
```

Но и никаких преимуществ по сравнению с массивовыми литералами она не дает. (В упражнении 2 описан тонкий и редко встречающийся случай употребления метода `of`.)

7.2. Свойство length и индексные свойства

У любого массива имеется свойство `'length'`, значением которого является целое число от 0 до $2^{32} - 1$. Свойства, ключами которых являются строки, эквивалентные неотрицательным целым числам, называются *индексными свойствами*. Например, массив

```
const names = ['Peter', 'Paul', 'Mary']
```

– это объект, имеющий свойство 'length' (со значением 3) и индексные свойства '0', '1', '2'. Напомним, что ключ свойства всегда является строкой.

Длина length всегда на единицу больше самого большого индекса:

```
const someNames = [ , 'Smith', , 'Jones'] // someNumbers.length равно 4
```

Длина корректируется, когда индексному свойству присваивается значение:

```
someNames[5] = 'Miller' // теперь someNames имеет длину 6
```

Можно изменить длину и вручную:

```
someNames.length = 100
```

Если длина уменьшается, то все элементы, индексы которых больше или равны новой длине, удаляются.

```
someNames.length = 4 // someNames[4] и последующие элементы удаляются
```

Не требуется, чтобы у массива были индексные свойства для всех индексов от 0 до length – 1. В стандарте ECMAScript употребляется термин *отсутствующие элементы* для обозначения лакун в последовательности индексов.

Чтобы выяснить, является ли элемент отсутствующим, можно воспользоваться оператором in:

```
'2' in someNames // false - свойства '2' нет
3 in someNames // true - свойство '3' есть
  // Отметим, что левый операнд преобразуется в строку
```

> **Примечание.** Массив может обладать свойствами, отличными от индексных. Так обычно присоединяют к массиву дополнительную информацию. Например, метод exec класса RegExp возвращает массив соответствий и имеет дополнительные свойства index и input.
>
> ```
> /([1-9]|1[0-2]):([0-5][0-9])([ap]m)/.exec('12:15pm')
>   // ['12:15pm', '12', '15', 'pm', index: 0, input: '12:15pm']
> ```

> **Предостережение.** Строка, содержащая отрицательное число, например '-1', является допустимым свойством, но не индексным.
>
> ```
> const squares = [0, 1, 4, 9]
> squares[-1] = 1 // [ 0, 1, 4, 9, '-1': 1 ]
> ```

7.3. Удаление и добавление элементов

Вызовы

```
let arr = [0, 1, 4, 9, 16, 25]
const deletedElement = arr.pop() // теперь arr содержит [0, 1, 4, 9, 16]
const newLength = arr.push(x) // теперь arr содержит [0, 1, 4, 9, 16, x]
```

соответственно удаляют и добавляют элемент в конец массива, в результате чего корректируется его длина.

> **Примечание.** Вместо того чтобы вызывать методы pop и push, можно было бы написать
>
> ```
> arr.length--
> arr[arr.length] = x
> ```
>
> Лично я предпочитаю pop и push, потому что они яснее отражают намерение.

Чтобы удалить или добавить первый элемент, нужно написать

```
arr = [0, 1, 4, 9, 16, 25]
const deletedElement = arr.shift() // теперь arr содержит [1, 4, 9, 16, 25]
const newLength = arr.unshift(x) // теперь arr содержит [x, 1, 4, 9, 16, 25]
```

Методы push и unshift могут добавить любое число элементов за один раз:

```
arr = [9]
arr.push(16, 25) // добавлены в конец 16, 25; теперь arr содержит [9, 16, 25]
arr.unshift(0, 1, 4) // добавлены в начало 0, 1, 4; теперь arr содержит [0, 1, 4, 9, 16, 25]
```

Для добавления элементов в середину служит метод splice:

```
const deletedElements = arr.splice(start, deleteCount, x1, x2, ...)
```

Сначала удаляется deleteCount элементов, начиная с позиции start. Затем указанные элементы вставляются, начиная с позиции start.

```
arr = [0, 1, 12, 24, 36]
const start = 2
// Заменить arr[start] и arr[start + 1]
arr.splice(start, 2, 16, 25) // теперь arr содержит [0, 1, 16, 25, 36]
// Добавить элементы, начиная с позиции start
arr.splice(start, 0, 4, 9) // теперь arr содержит [0, 1, 4, 9, 16, 25, 36]
// Удалить элементы в позициях start и start + 1
arr.splice(start, 2) // теперь arr содержит [0, 1, 16, 25, 36]
// Удалить все элементы, начиная с позиции start
arr.splice(start) // теперь arr содержит [0, 1]
```

Если start отрицательно, то позиция отсчитывается от *конца* массива (т. е. корректируется путем прибавления arr.length).

```
arr = [0, 1, 4, 16]
arr.splice(-1, 1, 9) // теперь arr содержит [0, 1, 4, 9]
```

Метод splice возвращает массив удаленных элементов.

```
arr = [1, 4, 9, 16]
const spliced = arr.splice(1, 2) // spliced равно [4, 9], arr равно [1, 16]
```

7.4. Прочие методы изменения массива

В этом разделе мы рассмотрим методы изменения объектов Array, помимо удаления и добавления элементов.

Метод fill перезаписывает существующие элементы новым значением:

```
arr.fill(value, start, end)
```

Метод copyWithin копирует на место существующих элементов другие элементы из того же массива:

```
arr.copyWithin(targetIndex, start, end)
```

В обоих методах start по умолчанию равно 0, а end – arr.length.

Если start, end или targetIndex отрицательны, то они отсчитываются от конца массива. Приведем несколько примеров:

```
let arr = [0, 1, 4, 9, 16, 25]
arr.copyWithin(0, 1) // теперь arr содержит [1, 4, 9, 16, 25, 25]
arr.copyWithin(1) // теперь arr содержит [1, 1, 4, 9, 16, 25]
arr.fill(7, 3, -1) // теперь arr содержит [1, 1, 4, 7, 7, 25]
```

Метод arr.reverse() обращает массив arr на месте:

```
arr = [0, 1, 4, 9, 16, 25]
arr.reverse() // теперь arr содержит [25, 16, 9, 4, 1, 0]
```

Вызов

```
arr.sort(comparisonFunction)
```

сортирует arr на месте. Функция сравнения принимает два элемента x, y и возвращает:

- отрицательное число, если x должен предшествовать y;
- положительное число, если x должен следовать за y;
- 0, если элементы неразличимы.

Вот, например, как можно отсортировать массив чисел:

```
arr = [0, 1, 16, 25, 4, 9]
arr.sort((x, y) => x - y) // теперь arr содержит [0, 1, 4, 9, 16, 25]
```

> **Предостережение.** Если функция сравнения не задана, то метод sort преобразует элементы в строки и сравнивает их (см. упражнение 5). Для чисел хуже такой функции сравнения ничего не придумаешь:
>
> ```
> arr = [0, 1, 4, 9, 16, 25]
> arr.sort() // теперь arr содержит [0, 1, 16, 25, 4, 9]
> ```

В табл. 7.1 сведены наиболее полезные методы класса Array.

Таблица 7.1. Полезные функции и методы класса `Array`

Имя	Описание
Функции	
`from(arraylike, f)`	Создает массив из любого объекта, обладающего свойствами `'length'`, `'0'`, `'1'` и т. д. Если функция `f` задана, то она применяется к каждому элементу
Изменяющие методы	
`pop()`, `shift()`	Удаляет и возвращает последний элемент
`push(value)`, `unshift(value)`	Добавляет `value` соответственно в конец или в начало массива и возвращает новую длину
`fill(value, start, end)`	Перезаписывает заданный диапазон значением `value`. Для этого и последующих методов действуют следующие правила, если явно не оговорено противное: если `start` или `end` отрицательно, то отсчет производится от конца массива. Диапазон включает `start` и не включает `end`. По умолчанию `start` равно `0`, а `end` – длине массива. Метод возвращает данный массив
`copyWithin(targetIndex, start, end)`	Копирует элементы из заданного диапазона в позиции, начиная с `targetIndex`
`reverse()`	Переставляет элементы данного массива в обратном порядке
`sort(comparisonFunction)`	Сортирует данный массив
`splice(start, deleteCount, values...)`	Удаляет и возвращает `deleteCount` элементов, начиная с позиции `start`, затем вставляет указанные элементы, начиная с позиции `start`
Неизменяющие методы	
`slice(start, end)`	Возвращает элементы в заданном диапазоне
`includes(target, start)`, `firstIndex(target, start)`, `lastIndex(target, start)`	Если массив содержит значение `target` в позиции `start` или после нее, то эти методы возвращают соответственно `true`, индекс первого или индекс последнего вхождения. В противном случае возвращается `false` или `-1`
`flat(k)`	Возвращает элементы данного массива, заменяя все массивы размерности ≤k их элементами. По умолчанию k равно `1`
`map(f)`, `flatMap(f)`, `forEach(f)`	Вызывает указанную функцию для каждого элемента массива и возвращает соответственно массив результатов, разглаженные результаты или `undefined`
`filter(f)`	Возвращает все элементы, для которых `f` возвращает значение, похожее на `true`
`findIndex(f)`, `find(f)`	Возвращает соответственно индекс или значение первого элемента, для которого `f` возвращает значение, похожее на `true`. Функции `f` передается три аргумента: элемент, индекс и массив
`every(f)`, `some(f)`	Возвращает `true`, если `f` возвращает значение, похожее на `true`, соответственно для всех или хотя бы одного элемента
`join(separator)`	Возвращает строку, содержащую все элементы, преобразованные в строки и разделенные строкой `separator` (по умолчанию `','`)

Для сортировки строк на естественном языке полезно использовать метод `localeCompare`:

```
const titles = . . .
titles.sort((s, t) => s.localeCompare(t))
```

Дополнительные сведения о локализованном сравнении см. в главе 8.

> **Примечание.** Начиная с 2019 года гарантируется, что метод `sort` осуществляет *устойчивую* сортировку. Это означает, что порядок неразличимых элементов не меняется. Предположим, к примеру, что имеется последовательность сообщений, уже отсортированная по дате. Если затем отсортировать ее по отправителю, то сообщения с одним и тем же отправителем останутся отсортированными по дате.

7.5. Порождение элементов

Ни один из описываемых ниже методов не изменяет массив, к которому применяется. Следующие методы порождают массивы, содержащие элементы из существующего массива. Вызов

```
arr.slice(start, end)
```

возвращает массив, содержащий элементы в заданном диапазоне. По умолчанию индекс `start` равен `0`, а `end` – `arr.length`. Вызов `arr.slice()` – то же самое, что `[...arr]`.

Метод `flat` разглаживает многомерные массивы. По умолчанию разглаживается один уровень. Результатом вызова

```
[[1, 2], [3, 4]].flat()
```

является массив

```
[1, 2, 3, 4]
```

В том маловероятном случае, когда имеется массив размерности больше 2, можно указать, сколько уровней разглаживать. В примере ниже производится разглаживание с трех уровней до одного:

```
[[[1, 2], [3, 4]], [[5, 6], [7, 8]]].flat(2) // [1, 2, 3, 4, 5, 6, 7, 8]
```

Вызов

```
arr.concat(arg1, arg2, . . .)
```

возвращает массив, начинающийся с `arr`, в конец которого добавлены аргументы. Но есть нюанс: аргументы-массивы разглаживаются.

```
const arr = [1, 2]
const arr2 = [5, 6]
const result = arr.concat(3, 4, arr2) // result равно [1, 2, 3, 4, 5, 6]
```

Поскольку теперь в массивовых литералах можно использовать оператор расширения, метод `concat` уже не очень полезен. Тот же результат можно получить проще:

```
const result = [...arr, 3, 4, ...arr2]
```

Но остался один случай, когда метод `concat` имеет смысл применять: для конкатенации последовательности элементов неизвестного типа с разглаживанием тех, которые являются массивами.

> **Примечание.** Разглаживание можно контролировать с помощью символа `isConcatSpreadable` (символы рассматриваются в главе 8).
> Если этот символ равен `false`, то массивы не разглаживаются:
>
> ```
> arr = [17, 29]
> arr[Symbol.isConcatSpreadable] = false
> [].concat(arr) // массив с одним элементом [17, 29]
> ```
>
> Если же символ равен `true`, то массивоподобные объекты разглаживаются:
>
> ```
> [].concat({ length: 2, [Symbol.isConcatSpreadable]: true,
> '0': 17, '1': 29 }) // массив с двумя элементами 17, 29
> ```

7.6. Поиск элементов

Следующие вызовы проверяют, присутствует ли указанное значение в массиве.

```
const found = arr.includes(target, start) // true или false
const firstIndex = arr.indexOf(target, start) // индекс первого вхождения или -1
const lastIndex = arr.lastIndexOf(target, start) // индекс последнего вхождения или -1
```

Параметр `target` должен быть строго равен элементу, для сравнения используется оператор `===`.

Поиск начинается с позиции `start`. Если `start` меньше `0`, то позиция отсчитывается от конца массива. Если параметр `start` опущен, то по умолчанию подразумевается `0`.

Если требуется найти значение, удовлетворяющее некоторому условию, то следует вызвать один из методов:

```
const firstIndex = arr.findIndex(conditionFunction)
const firstElement = arr.find(conditionFunction)
```

Вот, например, как можно найти первый отрицательный элемент в массиве:

```
const firstNegative = arr.find(x => x < 0)
```

В этом и следующих методах из этого раздела функция условия принимает три аргумента:

1) элемент массива;
2) индекс этого элемента;
3) весь массив.

Методы

```
arr.every(conditionFunction)
arr.some(conditionFunction)
```

возвращают `true`, если `conditionFunction(element, index, arr)` равно `true` соответственно для каждого или хотя бы одного элемента.

Например:

```
const atLeastOneNegative = arr.some(x => x < 0)
```

Метод `filter` возвращает все значения, удовлетворяющие заданному условию:

```
const negatives = [-1, 7, 2, -9].filter(x => x < 0) // [-1, -9]
```

7.7. Перебор всех элементов

Для посещения всех элементов массива можно использовать цикл `for of`, если нужно посетить элементы по порядку, или цикл `for in`, если нужно посетить все значения индекса.

```
for (const e of arr) {
  // Что-то сделать с элементом e
}
for (const i in arr) {
  // Что-то сделать с индексом i и элементом arr[i]
}
```

> **Примечание.** В цикле `for of` перебираются элементы с индексами от `0` до `length – 1`, и для отсутствующих элементов возвращается `undefined`. С другой стороны, в цикле `for in` посещаются только присутствующие ключи.
> Иными словами, цикл `for in` рассматривает массив как объект, а цикл `for of` – как итерируемый объект. (В главе 12 мы увидим, что итерируемые объекты – это последовательности значений без лакун.)

Если требуется посетить и индексы, и сами элементы, то воспользуйтесь итератором, возвращаемым методом `entries`. Он порождает массивы длины 2, каждый из которых содержит индекс и соответствующий элемент. В следующем цикле `for of` используется метод `entries` и деструктуризация:

```
for (const [index, element] of arr.entries())
  console.log(index, element)
```

> **Примечание.** Метод `entries` определен для всех структур данных в JavaScript, а не только для массивов. Существуют также методы `keys` и `values`, возвращающие итераторы, которые обходят только ключи и только значения коллекции. Они полезны при работе с обобщенными коллекциями. Если вы точно знаете, что работаете с массивом, то в этих методах нет необходимости.

Метод `arr.forEach(f)` вызывает `f(element, index, arr)` для каждого элемента массива, пропуская отсутствующие. Вызов

```
arr.forEach((element, index) => console.log(index, element))
```

эквивалентен циклу

```
for (const index in arr) console.log(index, arr[index])
```

Вместо того чтобы задавать действие для каждого элемента, часто бывает лучше преобразовать элементы и собрать результаты. Вызов `arr.map(f)` возвращает массив всех значений, возвращенных функцией `f(arr[index], index, arr)`:

```
[1, 7, 2, 9].map(x => x * x) // [1, 49, 4, 81]
[1, 7, 2, 9].map((x, i) => x * 10 ** i) // [1, 70, 200, 9000]
```

Рассмотрим функцию, которая возвращает массив значений:

```
function roots(x) {
    if (x < 0) {
      return [] // нет корней
    } else if (x === 0) {
    return [0] // один корень
    } else {
      return [Math.sqrt(x), -Math.sqrt(x)] // два корня
    }
}
```

Применив эту функцию к массиву входных данных, мы получим массив ответов, каждый из которых является массивом:

```
[-1, 0, 1, 4].map(roots) // [[], [0], [1, -1], [2, -2]]
```

Если вы хотите разгладить результаты, то вызовите `flat` вслед за `map` или сразу вызовите `flatMap` – это немного эффективнее:

```
[-1, 0, 1, 4].flatMap(roots) // [0, 1, -1, 2, -2]
```

Наконец, вызов `arr.join(separator)` преобразует все элементы массива в строки и соединяет их заданным разделителем `separator`. По умолчанию разделитель равен `','`.

```
[1,2,3,[4,5]].join(' and ') // '1 and 2 and 3 and 4,5'
```

> **Примечание.** Методы `forEach`, `map`, `filter`, `find`, `findIndex`, `some` и `every` (но не `sort` и `reduce`), а также функция `from` принимают необязательный аргумент после аргумента-функции:
>
> `arr.forEach(f, thisArg)`
>
> Аргумент `thisArg` становится для `f` параметром `this`. То есть для каждого индекса вызывается
>
> `thisArg.f(arr[index], index, arr)`
>
> Аргумент `thisArg` нужен, только если передается метод, а не свободная функция. В упражнении 4 показано, как избежать такой ситуации.

7.8. Разреженные массивы

Массив, в котором один или несколько элементов отсутствуют, называется *разреженным*. Разреженные массивы возникают в следующих ситуациях:

1) отсутствующие элементы в массивовом литерале:

```
const someNumbers = [ , 2, , 9] // Нет индексных свойств 0, 2
```

2) добавление элемента с индексом, большим или равным длине массива:

```
someNumbers[100] = 0 // нет индексных свойств 4-99
```

3) увеличение длины массива:

```
const bigEmptyArray = []
bigEmptyArray.length = 10000 // нет индексных свойств
```

4) удаление элемента:

```
delete someNumbers[1] // больше нет индексного свойства 1
```

Большинство методов в API массива пропускают отсутствующие элементы. Например, в случае `[ , 2, , 9].forEach(f)` функция `f` вызывается только два раза. Для отсутствующих элементов в позициях 0 и 2 функция не вызывается.

В разделе 7.1 мы видели, что `Array.from(arrayLike, f)` составляет исключение – `f` вызывается для каждого индекса.

Можно использовать функцию `Array.from` для замены отсутствующих элементов значением `undefined`:

```
Array.from([ , 2, , 9]) // [undefined, 2, undefined, 9]
```

Метод `join` преобразует отсутствующие элементы и элементы, равные `undefined`, в пустые строки:

```
[ , 2, undefined, 9].join(' and ') // ' and 2 and and 9'
```

Большинство методов, создающих новые массивы из существующих, оставляют отсутствующие элементы на месте. Например, `[ , 2, , 9].map(x => x * x)` дает `[ , 4, , 81]`.

Однако метод `sort` перемещает отсутствующие элементы в конец:

```
let someNumbers = [ , 2, , 9]
someNumbers.sort((x, y) => y - x) // теперь someNumbers содержит [9, 2, , , ]
```

(Внимательный читатель, вероятно, обратил внимание на четыре запятые. Последняя запятая завершающая. Если бы ее не было, то завершающей считалась бы предыдущая запятая, и в массиве был бы только один элемент `undefined`.)

Методы `filter`, `flat` и `flatMap` вообще пропускают отсутствующие элементы.

7.9. Редукция

В этом разделе обсуждается общий механизм вычисления одного значения по элементам массива. Механизм элегантный, но, честно говоря, необходимости в нем никогда не возникает – того же эффекта можно достичь с помощью простого цикла. Можете пропустить этот раздел, если он вам не интересен.

Метод `map` применяет унарную функцию ко всем элементам коллекции. Методы `reduce` и `reduceRight`, обсуждаемые в этом разделе, комбинируют элементы с помощью *бинарной* операции. Вызов `arr.reduce(op)` применяет операцию `op` к соседним элементам следующим образом:

```
            .
           .
          .
         op
        /  \
       op   arr[3]
      /  \
     op   arr[2]
    /  \
arr[0]   arr[1]
```

Вот, например, как вычисляется сумма элементов массива:

```
const arr = [1, 7, 2, 9]
const result = arr.reduce((x, y) => x + y) // ((1 + 7) + 2) + 9
```

А вот более интересный пример применения редукции для вычисления значения десятичного числа по массиву составляющих его цифр:

```
[1, 7, 2, 9].reduce((x, y) => 10 * x + y) // 1729
```

На диаграмме ниже показаны промежуточные результаты:

```
      1729
     /    \
    172    9
   /   \
  17    2
 /  \
1    7
```

В большинстве случаев полезно начинать вычисление с начального значения, отличного от первого элемента массива. Обращение к `arr.reduce(op, init)` вычисляет:

```
            .
           .
          .
         op
```

```
        /  \
       op   arr[2]
      /  \
     op   arr[1]
    /  \
 init   arr[0]
```

По сравнению с древовидной диаграммой `reduce` без начального значения, эта диаграмма более регулярна. Каждая операция комбинирует уже вычисленное значение (начиная с начального) с очередным элементом массива. Например:

```
[1, 7, 2, 9].reduce((accum, current) => accum - current, 0)
```

дает

```
0 - 1 - 7 - 2 - 9 = -19
```

Без начального значения мы получили бы 1 – 7 – 2 – 9, что не является суммой всех элементов с обратным знаком.

Если массив пуст, то возвращается само начальное значение. Например, если определить

```
const sum = arr => arr.reduce((accum, current) => accum + current, 0)
```

то сумма элементов пустого массива будет равна 0. Редукция пустого массива без начального значения возбуждает исключение.

Функция обратного вызова на самом деле принимает четыре аргумента:

1) вычисленное к этому моменту значение;
2) текущий элемент массива;
3) индекс текущего элемента;
4) массив целиком.

В следующем примере мы собираем позиции всех элементов, удовлетворяющих условию `condition`:

```
function findAll(arr, condition) {
  return arr.reduce((accum, current, currentIndex) =>
    condition(current) ? [...accum, currentIndex] : accum, [])
}

const odds = findAll([1, 7, 2, 9], x => x % 2 !== 0)
// [0, 1, 3], позиции всех нечетных элементов
```

Метод `reduceRight` начинает работу с конца массива и посещает элементы в обратном порядке:

```
            op
           /  \
          .   arr[0]
         .
        .
       /
```

```
        op
       /  \
      op   arr[n-2]
     /  \
  init   arr[n-1]
```

Например:

```
[1, 2, 3, 4].reduceRight((x, y) => [x, y], [])
```

равно

```
[[[[[], 4], 3], 2], 1]
```

> **Примечание.** Правая редукция в JavaScript похожа на правую свертку в языках, производных от Lisp, но порядок операндов обратный.

Свертку можно использовать вместо цикла. Пусть, например, требуется подсчитать частоты букв в строке. Один из способов – перебрать все буквы, обновляя по ходу дела объект:

```
const freq = {}
for (const c of 'Mississippi') {
  if (c in freq) {
    freq[c]++
  } else {
    freq[c] = 1
  }
}
```

Но можно подойти к задаче и по-другому. На каждом шаге комбинируется отображение частот и очередная буква, в результате чего получается новое отображение частот. Вот как выглядит редукция:

```
                        .
                       .
                      .
                     op
                    /  \
                   op  's'
                  /  \
                 op  'i'
                /  \
  пустое отображение 'M'
```

Что здесь такое `op`? Левый операнд – частично заполненное отображение частот, а правый операнд – очередная буква. Результатом должно быть пополненное отображение. Оно становится входными данными для следующего обращения к `op`, и конечным результатом является отображение, содержащее все счетчики. Код выглядит так:

```
[...'Mississippi'].reduce(
  (freq, c) => ({ ...freq, [c]: (c in freq ? freq[c] + 1 : 1) }),
  {})
```

В функции редукции создается новый объект, являющийся копией объекта `freq`. Затем либо создается новый ключ `c` и с ним ассоциируется значение 1, либо значение существующего ключа увеличивается на единицу.

Заметим, что при таком подходе нет никакого изменяющегося состояния. На каждом шаге вычисляется новый объект.

Любой цикл можно заменить обращением к `reduce`. Поместите все переменные, обновляемые в цикле, в объект и определите операцию, которая реализует один шаг цикла, порождая новый объект с обновленными переменными. Я не говорю, что это хорошая идея, но, возможно, вам интересно, что от циклов можно избавиться таким образом.

7.10. Отображения

В JavaScript API входит класс `Map`, реализующий классическую структуру данных отображение, т.е. коллекцию пар [ключ, значение].

Разумеется, любой объект в JavaScript и так является отображением, но у класса `Map` есть некоторые преимущества:

- ключи объектов должны быть строками или символами, а ключи `Map` могут быть любого типа;
- экземпляр `Map` запоминает порядок, в котором добавлялись элементы;
- в отличие от объектов, отображения не тянут за собой цепочку прототипов;
- количество записей в отображении можно получить с помощью свойства `size`.

Чтобы сконструировать отображение, необходим итерируемый объект, содержащий пары [*ключ*, *значение*]:

```
const weekdays = new Map(
  [["Mon", 0], ["Tue", 1], ["Wed", 2], ["Thu", 3], ["Fri", 4], ["Sat", 5], ["Sun", 6], ])
```

Можно вместо этого сконструировать пустое отображение и добавить записи позже:

```
const emptyMap = new Map()
```

Конструктор обязательно употреблять с ключевым словом `new`. API очень простой. Вызов

```
map.set(key, value)
```

добавляет запись и возвращает само отображение для сцепления вызовов:

```
map.set(key1, value1).set(key2, value2)
```

Чтобы удалить запись, вызовите

```
map.delete(key) // возвращает true, если ключ существовал, иначе false
```

Метод `clear` удаляет все записи.

Для проверки существования ключа вызовите

```
if (map.has(key)) . . .
```

Чтобы получить значение ключа, вызовите

```
const value = map.get(key) // Возвращает undefined, если ключа нет
```

Отображение является итерируемым объектом, который отдает пары [key, value]. Поэтому можно легко перебрать все записи в цикле for of:

```
for (const [key, value] of map) {
    console.log(key, value)
}
```

Или использовать метод forEach:

```
map.forEach((key, value) => {
    console.log(key, value)
})
```

Обход отображения производится в порядке вставки записей. Рассмотрим отображение:

```
const weekdays = new Map([['Mon', 0], ['Tue', 1], . . ., ['Sun', 6]])
```

И цикл for of, и метод forEach перебирают записи в порядке вставки.

> **Примечание.** В Java для посещения элементов в порядке вставки нужно было бы использовать класс LinkedHashMap. В JavaScript порядок вставки запоминается автоматически.

> **Примечание.** У отображений, как и у всех коллекций в JavaScript, имеются методы keys, values и entries, которые возвращают итераторы по ключам, значениям и парам ключ-значение. Если нужно обойти только ключи, можно воспользоваться циклом
>
> ```
> for (const key of map.keys()) . . .
> ```

В таких языках программирования, как Java и C++, есть выбор между отображениями на основе хеш-таблиц и деревьев и требуется предоставить функцию хеширования или сравнения. В JavaScript отображение всегда основано на хеш-таблицах и функцию хеширования выбирает сама система.

Функция хеширования для класса JavaScript Map совместима с *равенством ключей* – оператором === с дополнительной особенностью: все значения NaN равны. Хеш-значения выводятся из значений примитивных типов или ссылок на объекты.

Это хорошо, когда ключи являются строками или числами или вам повезло, и ключи можно сравнивать по тождественности. Например, отображение можно использовать, чтобы ассоциировать значения с узлами DOM. Это лучше, чем добавлять свойства прямо в объекты узлов.

Но будьте осторожны, когда ключами служат объекты. Разные объекты являются различными ключами, даже если их значения совпадают:

```
const map = new Map()
const key1 = new Date('1970-01-01T00:00:00.000Z')
const key2 = new Date('1970-01-01T00:00:00.000Z')
map.set(key1, 'Hello')
map.set(key2, 'Epoch') // теперь имеется две записи
```

Если это не то, что вам нужно, то рассмотрите возможность выбора других ключей, например строковых представлений дат в примере выше.

7.11. Множества

`Set` – структура данных, в которой не может быть дубликатов. Множество конструируется следующим образом:

```
const emptySet = new Set()
const setWithElements = new Set(iterable)
```

где объект `iterable` порождает элементы.

Как и в случае отображений, свойство `size` возвращает количество элементов.

API множеств еще проще, чем для отображений:

```
set.add(x)
  // добавляет элемент x, если его еще нет, и возвращает само множество для сцепления
  // методов
set.delete(x)
  // если x присутствует, то удаляет x и возвращает true, иначе возвращает false
set.has(x) // возвращат true, если x присутствует
set.clear() // удаляет все элементы
```

Перебрать все элементы множества можно в цикле `for of`:

```
for (const value of set) {
  console.log(value)
}
```

Или воспользоваться методом `forEach`:

```
set.forEach(value => {
  console.log(value)
})
```

Как и отображения, множества запоминают порядок вставки. Предположим, к примеру, что названия дней недели добавлялись по порядку:

```
const weekdays = new Set(['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'])
```

Тогда и цикл `for of`, и метод `forEach` будут обходить элементы в этом порядке.

> **Примечание.** Множество рассматривается как отображение, составленное из пар [значение, значение]. Методы keys и values возвращают итератор по значениям, а метод entries – итератор по парам [значение, значение]. Все эти методы не нужны, когда заведомо известно, что вы работаете с множеством. Они предназначены для использования в обобщенном коде, работающем с произвольными коллекциями.

Как и отображения, множества реализованы в виде хеш-таблиц с предопределенной функцией хеширования. Элементы множества считаются одинаковыми, если они принадлежат примитивному типу и имеют одинаковые значения или являются одной и той же ссылкой на объект. Кроме того, все значения NaN равны друг другу.

7.12. Слабые отображения и множества

Важный случай использования отображений и множеств в JavaScript – присоединение свойств к узлам DOM. Допустим, требуется классифицировать некоторые узлы, отнеся их к категориям успех, в работе и ошибка. Можно было бы присоединить свойства непосредственно к узлам:

```
node.outcome = 'success'
```

В общем-то, такой код работает, но он хрупкий. У узлов DOM очень много свойств, и беда подстерегает того, кто выберет свойство, совпадающее с тем, что уже существует сейчас или появится в будущей версии DOM API.

Надежнее использовать отображение:

```
const outcome = new Map()
...
outcome.set(node, 'success')
```

Узлы DOM появляются и исчезают. Если какой-то узел больше не нужен, им должен заняться сборщик мусора. Однако если ссылка на узел хранится в отображении outcome, то эта ссылка не даст объекту умереть.

Тут-то и приходят на помощь *слабые отображения*. Если ключ слабого отображения – единственная ссылка на объект, то сборщик мусора не считает этот объект живым.

Просто разместите дополнительные свойства в слабом отображении:

```
const outcome = new WeakMap()
```

У слабых отображений нет методов обхода, и они не являются итерируемыми объектами. У них есть только методы set, delete, has и get. Этого достаточно, чтобы задать свойство и проверить, есть ли свойство у заданного объекта.

Если отслеживаемое свойство бинарное, то можно воспользоваться классом WeakSet вместо WeakMap. У него свойств еще меньше: set, delete и has.

Ключами слабых отображений и элементами слабых множеств могут быть только объекты, но не значения примитивных типов.

7.13. Типизированные массивы

В массивах JavaScript хранятся последовательности элементов любой природы, и некоторые элементы могут отсутствовать. Если вам всего-то и нужно, что сохранить последовательность чисел или байты изображения, то массив общего вида неэффективен.

Для эффективного хранения последовательностей чисел одного и того же типа можно использовать *типизированный массив*. Имеются следующие типы массивов:

```
Int8Array
Uint8Array
Uint8ClampedArray
Int16Array
Uint16Array
Int32Array
Uint32Array
Float32Array
Float64Array
```

Все элементы имеют заданный тип. Например, в массиве типа `Int16Array` хранятся 16-разрядные целые в диапазоне от –32 768 до 32 767. Префикс `Uint` обозначает целые без знака. В массиве типа `UInt16Array` хранятся целые числа в диапазоне от 0 до 65 535.

При конструировании массива задается его длина. Изменить ее впоследствии невозможно.

```
const iarr = new Int32Array(1024)
```

Сразу после конструирования все элементы массива равны нулю.

Не существует типизированных массивовых литералов, но в каждом классе типизированного массива имеется функция `of` для конструирования экземпляра по заданным значениям:

```
const farr = Float32Array.of(1, 0.5, 0.25, 0.125, 0.0625, 0.03215, 0.015625)
```

Как и в случае массивов, имеется функция `from`, которая получает элементы из любого итерируемого объекта и необязательную функцию отображения:

```
const uarr = Uint32Array.from(farr, x => 1 / x)
  // Массив типа Uint32Array с элементами [1, 2, 4, 8, 16, 32, 64]
```

Присваивание элементу с числовым индексом, не являющимся целым числом от 0 до `length – 1`, ничего не делает. Но, как и в обычных массивах, можно задавать другие свойства:

```
farr[-1] = 2 // ничего не изменилось
farr[0.5] = 1.414214 // ничего не изменилось
farr.lucky = true // установлено свойство lucky
```

Когда элементу целочисленного массива присваивается число, дробная часть отбрасывается. После этого число приводится к диапазону, соответствующему типу массива. Рассмотрим пример:

```
iarr[0] = 40000.25 // в iarr[0] записывается -25536
```

Используется только целая часть. Поскольку `40000` не помещается в диапазон 32-разрядных целых чисел, берутся последние 32 бита, им соответствует число –25 536.

Исключением из этого процесса приведения является тип `Uint8ClampedArray` – значения, выходящие за пределы диапазона, заменяются на 0 или 255, а нецелые значения округляются до ближайшего целого.

Тип `Uint8ClampedArray` предназначен для использования совместно с изображениями на HTML-холсте. Метод контекста холста `getImageData` возвращает объект, свойство `data` которого имеет тип `Uint8ClampedArray` и содержит RGBA-значения пикселей в прямоугольнике на холсте:

```
const canvas = document.getElementById('canvas')
const ctx = canvas.getContext('2d')
ctx.drawImage(img, 0, 0)
let imgdata = ctx.getImageData(0, 0, canvas.width, canvas.height)
let rgba = imgdata.data // массив типа Uint8ClampedArray
```

Рис. 7.1 ❖ Содержимое холста инвертируется в ответ на щелчок мышью

В коде, сопровождающем книгу, имеется пример программы, которая инвертирует цвета на холсте по щелчку мышью, – см. рис. 7.1.

```
canvas.addEventListener('click', event => {
    for (let i = 0; i < rgba.length; i++) {
      if (i % 4 != 3) rgba[i] = 255 - rgba[i]
    }
  ctx.putImageData(imgdata, 0, 0)
})
```

У типизированных массивов есть все методы обычных массивов, кроме следующих:

- `push`, `pop`, `shift`, `unshift` – размер типизированного массива нельзя изменять;
- `flat`, `flatMap` – элементами типизированного массива не могут быть массивы;
- `concat` – используйте вместо этого `set`.

Есть также два метода, отсутствующих у обычных массивов. Метод `set` копирует все значения из массива или типизированного массива в позиции, начиная с заданного смещения:

```
targetTypedArray.set(source, offset)
```

По умолчанию смещение равно нулю. Исходный массив `source` должен целиком умещаться в целевой. Если сумма смещения и длины исходного массива превышает длину целевого массива, то возбуждается исключение `RangeError`. (Следовательно, этот метод нельзя использовать для сдвига элементов типизированного массива.)

Метод `subarray` возвращает представление поддиапазона элементов:

```
const sub = iarr.subarray(16, 32)
```

Если конечный индекс опущен, то подразумевается длина массива, а опущенный начальный индекс считается равным нулю.

На первый взгляд, это то же самое, что метод `slice`, но есть одно важное различие. У массива и подмассива одни и те же элементы. Модификация одного отражается на другом.

```
sub[0] = 1024 // сейчас iarr[16] также равно 1024
```

7.14. Буферные массивы

Буферный массив (array buffer) – это непрерывная последовательность байтов, в которой можно хранить данные из файла, поток данных, изображение и т. д. Данные из типизированных массивов также хранятся в буферных массивах.

Целый ряд API для работы с вебом (в т. ч. файловый API, `XMLHttpRequest` и веб-сокеты) возвращают буферные массивы. Вы и сами можете сконструировать буферный массив с заданным числом байтов:

```
const buf = new ArrayBuffer(1024 * 2)
```

Обычно двоичные данные в буферном массиве имеют сложную структуру, как, скажем, изображение или звуковой файл. Тогда используйте класс `DataView`, чтобы изучить хранящиеся данные:

```
const view = new DataView(buf)
```

Для чтения данных с заданным смещением служат методы `DataView` `getInt8`, `getInt16`, `getInt32`, `getUInt8`, `getUInt16`, `getUInt32`, `getFloat32`, `getFloat64`:

```
const littleEndian = true // false или опустить, если порядок байтов прямой
const value = view.getUint32(offset, littleEndian)
```

Для записи данных служит метод `set`:

```
view.setUint32(offset, newValue, littleEndian)
```

> **Примечание.** Существует два способа хранения двоичных данных в виде последовательности байтов: прямой (big-endian) и обратный (little-endian). Рассмотрим 16-разрядное число `0x2122`. При прямом порядке байтов старший байт записывается первым: сначала `0x21`, потом `0x22`. При обратном порядке все наоборот: `0x22 0x21`. В большинстве современных процессоров используется обратный порядок байтов, но во многих распространенных форматах файлов (например, PNG и JPEG) порядок байтов прямой.
> Термины «big-endian» (букв. тупоконечный) и «little-endian» (остроконечный) заимствованы из сатирического романа Джонатана Свифта «Путешествия Гулливера» и относятся к двум концам яйца.

В буфере типизированного массива всегда используется порядок байтов платформы, на которой исполняется программа. Если буфер целиком занят массивом и известно, что порядок байтов в нем совпадает с платформенным порядком байтов, то можно построить типизированный массив по содержимому буфера:

```
const arr = new Uint16Array(buf) // массив из 1024 Uint16, хранящийся в buf
```

Упражнения

1. Напишите функцию, которая работает в точности так же, как функция `from` из класса `Array`. Особое внимание обращайте на отсутствующие элементы. Что происходит с объектами, для которых числовые значения ключей больше или равны свойству `length`? А со свойствами, отличными от индексных?
2. Функция `Array.of` проектировалась для очень узкого использования: передачи в качестве «коллектора» функции, которая порождает последовательность значений и отправляет их в какое-то место назначения – быть может, печатает, суммирует или собирает в массив. Напишите такую функцию:

   ```
   mapCollect(values, f, collector)
   ```

   Эта функция должна применять `f` к каждому значению, а затем отправлять результат коллектору – функции с переменным числом аргументов. Возвращается результат коллектора.
   Объясните, в чем преимущество использования `Array.of` по сравнению с `Array` (т. е. `(...elements) => new Array(...elements)`) в этом контексте.
3. У массива могут быть свойства, числовые значения которых – отрицательные целые числа, например `'-1'`. Влияют ли они на длину? Как можно обойти их по порядку?

4. Погуглите «JavaScript forEach thisArg», чтобы найти в блогах статьи, в которых объясняется смысл параметра `thisArg` метода `forEach`. Перепишите приведенные в них примеры без использования параметра `thisArg`. Если встретите вызов вида

   ```
   arr.forEach(function() { . . . this.something() . . . }, thisArg)
   ```

   в котором `thisArg` равно `this`, замените эту функцию стрелочной. В противном случае замените внутренний `this` тем, чему равен `thisArg`. Если вызов имеет вид

   ```
   arr.forEach(method, thisArg)
   ```

   то используйте стрелочную функцию, вызывающую `thisArg.method(...)`. Можете ли вы придумать ситуацию, в которой `thisArg` действительно необходим?
5. Если не передать функцию сравнения методу `sort` класса `Array`, то элементы преобразуются в строки и лексикографически сравниваются составляющие их кодовые единицы UTF-16. Почему эта идея никуда не годится? Приведите примеры массивов целых чисел или объектов, для которых результаты такой сортировки бесполезны. А как насчет символов с кодами больше `\u{FFFF}`?
6. Предположим, что объект, представляющий сообщение, имеет свойства для даты и отправителя. Отсортируйте массив сообщений сначала по дате, а затем по отправителю. Убедитесь, что метод `sort` действительно устойчивый: сообщения с одинаковым отправителем остаются отсортированными по дате после второй сортировки.
7. Предположим, что объект, представляющий человека, имеет свойства для имени и фамилии. Напишите функцию сравнения, которая сравнивает фамилии, а в случае совпадения сравнивает имена.
8. Напишите функцию сравнения, которая сравнивает две строки по кодовым точкам Юникода, а не по кодовым единицам UTF-16 .
9. Напишите функцию, которая возвращает все позиции заданного значения в массиве. Например, `indexOf(arr, 0)` возвращает все такие индексы `i`, что `arr[i]` равно нулю. Воспользуйтесь методами `map` и `filter`.
10. Напишите функцию, которая возвращает все позиции, в которых заданная функция принимает значение `true`. Например, `indexOf(arr, x => x > 0)` возвращает все такие индексы `i`, что `arr[i]` положительно.
11. Вычислите размах (т. е. разность между максимальным и минимальным значениями) массива с помощью метода `reduce`.
12. Пусть дан массив функций $[f_1, f_2, \ldots, f_n]$. Постройте композицию `x => f1(f2(... (fn(x)) ... ))` с помощью метода `reduceRight`.
13. Напишите функции `map`, `filter`, `forEach`, `some`, `every` для множеств.
14. Напишите функции `union(set1, set2)`, `intersection(set1, set2)`, `difference(set1, set2)`, которые возвращают объединение, пересечение и разность множеств, не изменяя аргументов.
15. Напишите функцию, которая конструирует отображение `Map` из объекта, так чтобы можно было легко построить отображение следующим образом: `toMap({ Monday: 1, Tuesday: 2, ... })`.

16. Предположим, что имеется отображение, ключами которого являются объекты, представляющие точки, вида `{ x: ..., y: ... }`. Что нехорошего может случиться при выполнении запросов типа `map.get({x: 0, y: 0 })`? Как можно это предотвратить?
17. Покажите, что слабые множества работают именно так, как обещано. Запустите Node.js с флагом `--expose-gc`. Вызовите метод `process.memoryUsage()`, чтобы узнать, какая часть кучи использована. Выделите память для объекта:

    ```
    let fred = { name: 'Fred', image: new Int8Array(1024*1024) }
    ```

    Убедитесь, что свободное место в куче уменьшилось примерно на один мегабайт. Присвойте переменной `fred` значение `null`, запустите сборщик мусора, вызвав метод `global.gc()`, и проверьте, убран ли объект в мусор. Повторите эти действия, вставив объект в слабое множество. Убедитесь, что слабое множество допускает уборку объекта в мусор. Повторите то же самое с обычным множеством и убедитесь, что сборщик мусора не убирает объект.
18. Напишите функцию, определяющую порядок байтов на данной платформе. Воспользуйтесь буферным массивом и рассмотрите его как представление данных и как типизированный массив.

Глава 8

Интернационализация

Окружающий нас мир велик, и хочется надеяться, что многие его обитатели заинтересуются написанной вами программой. Некоторые программисты думают, что для интернационализации приложения достаточно поддержать Юникод и перевести сообщения пользовательского интерфейса. Однако, как мы вскоре увидим, интернационализация программ к этому отнюдь не сводится. Даты, время, денежные единицы и даже числа форматируются поразному в разных странах. В этой главе мы увидим, как использовать встроенные в JavaScript средства интернационализации, чтобы программа могла отображать и принимать информацию так, как ожидают пользователи, где бы они ни находились.

8.1. Понятие локали

Когда смотришь на приложение, рассчитанное на интернациональную аудиторию, в глаза прежде всего бросается язык. Но есть много и других, более тонких различий, например числа форматируются по-разному в разных странах. Число

```
123,456.78
```

должно отображаться в виде

```
123.456,78
```

для пользователя из Германии, т. е. роли десятичной точки и запятой в качестве разделителя меняются. В других локалях пользователи предпочитают иное написание цифр. Вот как то же число выглядит в тайской письменности:

```
๑๒๓,๔๕๖.๗๘
```

Аналогичные вариации имеют место для дат. В США даты отображаются в виде месяц/день/год, в Германии используется не столь причудливый порядок – день/месяц/год, а в Китае еще более разумный – год/месяц/день. Таким образом, дата, которую американец запишет в виде

```
3/22/61
```

немцу должна быть предъявлена в виде

```
22.03.1961
```

Когда названия месяцев записываются словами, языковые различия становятся еще более наглядными. Английское

```
March 22, 1961
```

для немца будет выглядеть как

```
22. März 1961
```

а для китайца как

```
1961年月22日
```

Локаль определяет язык и местоположение пользователя и позволяет средствам форматирования принимать во внимание предпочтения пользователей. В следующих разделах показано, как задается локаль и как управлять ее параметрами в программе на JavaScript.

8.2. Задание локали

Локаль состоит из пяти компонент.

1. Язык, который обозначается двумя или тремя строчными буквами, например `en` (английский), `de` (немецкий) или `zh` (китайский). В табл. 8.1 перечислены наиболее часто встречающиеся коды.

Таблица 8.1. Наиболее распространенные коды языков

Язык	Код	Язык	Код
Китайский	zh	Японский	ja
Датский	da	Корейский	ko
Голландский	du	Норвежский	no
Английский	en	Португальский	pt
Французский	fr	Испанский	es
Финский	fi	Шведский	sv
Итальянский	it	Турецкий	tr

2. Факультативная письменность, обозначаемая четырьмя буквами, первая из которых заглавная, например `Latn` (латиница), `Cyrl` (кириллица) или `Hans` (упрощенные китайские иероглифы). Это может быть полезно, поскольку в некоторых языках, например сербском, употребляется как латиница, так и кириллица, а некоторые китайские пользователи предпочитают традиционные иероглифы упрощенным.
3. Факультативные страна или регион, обозначаемые двумя заглавными буквами или тремя цифрами, например `US` (США) или `CH` (Швейцария).

В табл. 8.2 перечислены наиболее употребительные коды.

Таблица 8.2. Коды некоторых стран

Страна	Код	Страна	Код
Австрия	AT	Япония	JP
Бельгия	BE	Корея	KR
Канада	CA	Нидерланды	NL
Китай	CN	Португалия	PT
Финляндия	FI	Испания	ES
Германия	DE	Швеция	SE
Великобритания	GB	Швейцария	CH
Греция	GR	Тайвань	TW
Ирландия	IE	Турция	TR
Италия	IT	США	US

4. Факультативный вариант. В настоящее время варианты используются редко. Когда-то существовал нюнорский (Nynorsk) вариант норвежского языка, но сейчас для него употребляется другой код языка, nn. А существовавшие ранее варианты для традиционного японского календаря или тайских числительных теперь оформляются с помощью расширений (см. ниже).
5. Факультативное расширение. Расширения описывают локальные предпочтения для календарей (например, японского), чисел (тайских вместо европейских) и т. д. Некоторые расширения специфицированы в стандарте Юникода. Имена этих расширений начинаются префиксом `u-` и двухбуквенным кодом, определяющим, к чему относится расширение: к календарю (`ca`), числам (`nu`) и т. д. Например, расширение `u-nu-thai` описывает использование тайских числительных. Имена нестандартизованных расширений должны начинаться префиксом `x-`, например `x-java`.

Правила оформления локалей изложены в меморандуме BCP 47 (Best Current Practices – лучшие современные практики) технической комиссии интернета (http://tools.ietf.org/html/bcp47). Более понятное краткое изложение имеется в статье по адресу www.w3.org/International/articles/language-tags.

> **Примечание.** Коды языков и стран не всегда очевидны, потому что некоторые основаны на местном написании. По-немецки German пишется «Deutsch», а по-китайски Chinese – zhongwen, отсюда коды de и zh. А код Швейцарии CH произведен от латинского названия Швейцарской конфедерации – «Confoederatio Helvetica».

Локали описываются тегами – написанными через дефис строками, включающими элементы имени, например `'en-US'`.

Немецкая локаль в Германии обозначается `'de-DE'`. А в Швейцарии четыре официальных языка (немецкий, французский, итальянский и ретороманский), поэтому немецкоговорящий швейцарец пользуется локалью `'de-CH'`. В этой локали закодированы правила немецкого языка, но денежные величины выражены в швейцарских франках, а не в евро.

Тег локали передается чувствительным к локали функциям, например:

```
const newYearsEve = new Date(1999, 11, 31, 23, 59)
newYearsEve.toLocaleString('de') // возвращается строка '31.12.1999 23:59:00'
```

Вместо одного тега локали можно передать массив в порядке убывания приоритетов: `['de-CH', 'de', 'en']`. В таком случае чувствительный к локали метод выберет первую локаль, которую способен поддержать.

В объекте, передаваемом после тега локали, можно задать дополнительные параметры:

```
newYearsEve.toLocaleString('de', { timeZone: 'Asia/Tokyo' })
  // Представлять дату в указанном часовом поясе, например '1.1.2000, 07:59:00'
```

Если локаль и параметры опущены, то берется локаль по умолчанию без параметров. Чтобы использовать локаль по умолчанию с параметрами, следует передать пустой массив локалей:

```
newYearsEve.toLocaleString([], { timeZone: 'Asia/Tokyo' })
```

> **Примечание.** Метод `toLocaleString` определен в объекте `Object`. Его можно переопределить в любом классе (см. упражнение 1).

8.3. Форматирование чисел

Для форматирования чисел вызовите метод `toLocaleString` класса `Number` и передайте ему тег локали в качестве аргумента:

```
let number = 123456.78
let result = number.toLocaleString('de') // '123,456.78'
```

Вместо этого можно сконструировать экземпляр класса `Intl.NumberFormat` и вызвать его метод `format`:

```
let formatter = new Intl.NumberFormat('de')
result = formatter.format(number) // '123,456.78'
```

В том маловероятном случае, если потребуется детализировать этот результат, можете воспользоваться методом `formatToParts`, который возвращает массив частей. Например, результатом вызова `formatter.formatToParts(number)` является такой массив:

```
[ { type: 'integer', value: '123' },
  { type: 'group', value: ',' },
  { type: 'integer', value: '456' },
  { type: 'decimal', value: '.' },
  { type: 'fraction', value: '78' } ]
```

Для любого чувствительного к локали метода важно знать, какие расширения локали и параметры он поддерживает. В табл. 8.3 эта информация приведена для метода `toLocaleString` класса `Number` и метода `format` класса `Intl.NumberFormat`.

Напомним, что имена расширений локали начинаются префиксом `u`. Метод `format` распознает расширения `u-nu`, например:

```
number.toLocaleString('th-u-nu-thai')

new Intl.NumberFormat('th-u-nu-thai').format(number)
  // Оба вызова возвращают '๑๒๓,๔๕๖.๗๘'
```

Параметры передаются во втором аргументе:

```
number.toLocaleString('de', { style: 'currency', currency: 'EUR' })
formatter = new Intl.NumberFormat('de', { style: 'currency', currency: 'EUR' })
formatter.format(number)
  // Оба вызова возвращают '123.456,78 €'
```

Как видим, если требуется многократно выполнять сложное форматирование, то имеет смысл сконструировать объект форматирования:

В упражнении 2 вы научитесь использовать различные параметры.

> **Примечание.** В предложении, находящемся на третьей стадии рассмотрения, описаны дополнительные параметры форматирования для единиц измерения (`'299,792,458 m/s'`), научной нотации (`'6.022E23'`) и компактной записи десятичных числительных (`'8.1 billion'`).

> **Предостережение.** К сожалению, в настоящее время не существует стандартного способа разбора локализованных чисел с разделителями групп или с цифрами, отличными от 0–9.

Таблица 8.3. Параметры метода `toLocaleString` для чисел и конструктора `Intl.NumberFormat`

Имя	Значение
Расширения тега локали	
`nu` (система написания чисел)	`latn`, `arab`, `thai`, ...
Параметры	
`style`	`decimal` (по умолчанию), `currency`, `percent`
`currency`	Код валюты в стандарте ISO 4217, например `USD` или `EUR`. Необходим для задания стиля денежных величин
`currencyDisplay`	`symbol` (€, по умолчанию), `code` (EUR), `name` (Euro)
`useGrouping`	`true` (по умолчанию), если используются разделители группы
`minimumIntegerDigits`, `minimumFractionDigits`, `maximumFractionDigits`, `minimumSignificantDigits`, `maximumSignificantDigits`	Ограничения на количество цифр до или после десятичного разделителя и на общее количество цифр

8.4. Локализация даты и времени

От локали зависят многие аспекты форматирования даты и времени.

1. Названия месяцев и дней недели должны быть записаны на местном языке.
2. Порядок следования года, месяца и дня зависит от страны.
3. Не всегда даты записываются в григорианском календаре.
4. Следует принимать во внимание часовой пояс.

В следующих разделах мы увидим, как локализовать объекты `Date`, диапазоны дат и относительные даты (например, «через 3 дня»).

8.4.1. Форматирование объектов Date

Имея объект класса `Date`, мы можем отформатировать только дату, только время или то и другое:

```
const newYearsEve = new Date(1999, 11, 31, 23, 59)
newYearsEve.toLocaleDateString('de') // '31.12.1999'
newYearsEve.toLocaleTimeString('de') // '23:59:00'
newYearsEve.toLocaleString('de') // '31.12.1999, 23:59:00'
```

Как и в случае чисел, можно сконструировать объект форматирования для данной локали и вызвать его метод `format`:

```
const germanDateTimeFormatter = new Intl.DateTimeFormat('de')
germanDateTimeFormatter.format(newYearsEve) // '31.12.1999'
```

Можно задать параметры, управляющие форматированием каждой части:

```
newYearsEve.toLocaleDateString('en', {
  year: 'numeric',
  month: 'short',
  day: 'numeric',
}) // 'Dec 31, 1999'

new Intl.DateTimeFormat('de', {
  hour: 'numeric',
  minute: '2-digit'
}).format(newYearsEve) // '23:59'
```

Однако это громоздкий и не слишком логичный подход. Ведь формат каждой части и даже вопрос о том, какие части включать, зависит от локали. Спецификация ECMAScript предписывает запутанный алгоритм подбора формата для заданной локали и параметр `formatMatcher`, который позволяет выбрать между алгоритмом из спецификации и потенциально лучшим. Такая сложность, видимо, навела авторов спецификации на мысль, что они пошли по ложному пути. Предложение по исправлению этой оплошности сейчас находится на третьей стадии рассмотрения. Нам нужно задать только желаемый стиль даты и времени (`full`, `long`, `medium` или `short`). А объект форматирования выберет подходящие для локали поля и форматы:

```
newYearsEve.toLocaleDateString('en', { dateStyle: 'medium' })
  // 'Dec 31, 1999'
newYearsEve.toLocaleDateString('de', { dateStyle: 'medium' })
  // '31.12.1999'
```

В табл. 8.4 показаны все расширения и параметры тегов локали.

Таблица 8.4. Параметры форматирования дат

Имя	Значение
Расширения тега локали	
nu (система написания чисел)	latn, arab, thai
ca (календарь)	gregory, hebrew, buddhist, …
hc (часовой цикл)	h11, h12, h23, h24
Параметры	
timeZone	UTC, Europe/Berlin, … (по умолчанию: местное время)
dateStyle, timeStyle (на третьей стадии рассмотрения)	full, long, medium, short. Если можете обойтись этим, избегайте перечисленных ниже параметров
hour12	true, false (использовать ли 12-часовой формат времени; значение по умолчанию зависит от локали)
hourCycle	h11, h12, h23, h24
year, day, hour, minute, second	2-digit (09), numeric (9), narrow (S), short (Sep), long (September)
weekday, era	long, short, narrow
timeZoneName	short (GMT+9), long (Japan Standard Time)
formatMatcher	basic (стандартный алгоритм сравнения запрошенного формата с предлагаемыми локалью), best fit (по умолчанию, потенциально лучшая реализация, предлагаемая исполняющей средой JavaScript)

8.4.2. Диапазоны

Метод formatRange класса Intl.DateTimeFormat форматирует диапазон между двумя датами с максимально возможной точностью:

```
const christmas = new Date(1999, 11, 24)
const newYearsDay = new Date(2000, 0, 1)
const formatter = new Intl.DateTimeFormat('en', { dateStyle: 'long' })
formatter.formatRange(christmas, newYearsEve) // 'December 24 – 31, 1999'
formatter.formatRange(newYearsEve, newYearsDay) // 'December 31, 1999 – January 1, 2000'
```

8.4.3. Относительное время

Класс Intl.RelativeTimeFormat порождает выражения вида «вчера» или «через 3 дня».

```
new Intl.RelativeTimeFormat('en', { numeric: 'auto'}).format(-1, 'day') // 'yesterday'
new Intl.RelativeTimeFormat('fr').format(3, 'hours') // 'dans 3 heures'
```

Метод `format` принимает два аргумента: количество и единицу времени. Единица времени может принимать следующие значения: `year`, `quarter`, `month`, `week`, `day`, `hour`, `minute`, `second`. Формы множественного числа, например `years`, также допустимы.

Можно задать следующие дополнительные параметры:

- `numeric`: `always` (1 день назад, по умолчанию), `auto` (вчера);
- `style`: `long`, `short`, `narrow`.

8.4.4. Форматирование с точностью до отдельных частей

Как и объекты форматирования чисел, классы `Intl.DateTimeFormat` и `Intl.RelativeTimeFormat` располагают методом `formatToParts`, который порождает массивы объектов, описывающих отдельные части отформатированного результата. Приведем два примера.

Вызов

```
new Intl.RelativeTimeFormat('fr').formatToParts(3, 'hours')
```

возвращает массив

```
[
  { type: 'literal', value: 'dans '},
  { type: 'integer', value: '3', unit: 'hour' },
  { type: 'literal', value: ' heures'}
]
```

Вызов

```
Intl.DateTimeFormat('en',
{
  dateStyle: 'long',
  timeStyle: 'short'
}).formatToParts(newYearsEve)
```

возвращает массив из 11 элементов, описывающих части строки `'December 31, 1999 at 11:59 PM'`, а именно:

```
{ type: 'month', value: 'December' },
{ type: 'literal', value: ' ' },
{ type: 'day', value: '31' },
```

и так далее.

8.5. Порядок следования

В JavaScript строки можно сравнивать с помощью операторов `<`, `<=`, `>` и `>=`. К сожалению, при взаимодействии с человеком эти операторы не слишком

полезны. Они приводят к абсурдным результатам, даже если ограничиться английским языком. Например, следующие пять строк упорядочены с помощью оператора <:

```
Athens
Zulu
able
zebra
Ångström
```

При построении словаря хотелось бы считать строчные и заглавные буквы эквивалентными, а диакритические знаки не принимать во внимание. Для англоговорящих этот список слов следовало бы расположить в порядке:

```
able
Ångström
Athens
zebra
Zulu
```

Однако пользователю из Швеции такой порядок не понравится. В шведском языке буква Å отличается от A и располагается *после* буквы Z!

То есть швед хотел бы расположить эти слова в таком порядке:

```
able
Athens
zebra
Zulu
Ångström
```

При сортировке строк, полученных от человека, всегда следует использовать сравнение с учетом локали.

Проще всего это сделать с помощью метода `localeCompare` класса `String`, передав локаль в качестве второго аргумента:

```
const words = ['Alpha', 'Ångström', 'Zulu', 'able', 'zebra']
words.sort((x, y) => x.localeCompare(y, 'en'))
  // слова расположены так: ['able', 'Alpha', 'Ångström', 'zebra', 'Zulu']
```

Или можно сконструировать объект сравнения:

```
const swedishCollator = new Intl.Collator('sv')
```

а затем передать функцию `compare` этого объекта методу `Array.sort`:

```
words.sort(swedishCollator.compare)
  // а теперь так: ['able', 'Alpha', 'zebra', 'Zulu', 'Ångström']
```

В табл. 8.5 перечислены расширения и параметры, поддерживаемые методом `localeCompare` и конструктором `Intl.Collator`.

Полезным расширением является числовая сортировка, когда подстроки, состоящие только из цифр, сортируются в порядке возрастания:

```
const parts = ['part1', 'part10', 'part2', 'part9']
parts.sort((x, y) => x.localeCompare(y, 'en-u-kn-true'))
  // Массив parts отсортирован так: ['part1', 'part2', 'part9', 'part10']
```

Многие другие конструкции имеют ограниченное применение. Например, в немецких телефонных книгах (но не в словарях) буква Ö считается эквивалентной буквосочетанию Oe. Следующий вызов не изменяет массив:

```
['Österreich', 'Offenbach'].sort((x, y) => x.localeCompare(y, 'de-u-co-phonebk'))
```

Таблица 8.5. Сравнение строк с помощью метода localeCompare и конструктора Intl.Collator

Имя	Значение
Расширения тега локали	
co (порядок следования)	phonebook, phonetic, reformed, pinyin, …
kn (числовое сравнение)	true ('1' < '2' < '10'), false (по умолчанию)
kf (какой регистр сначала)	upper, lower, false (по умолчанию)
Параметры	
sensitivity	base (a = A = Å), accent (a = A ≠ Å), case (a ≠ A = Å), variant (a, A, Å различны; по умолчанию)
ignorePunctuation	true, false (по умолчанию)
numeric, caseFirst	true, false (по умолчанию) – см. kn, kf выше
usage	sort (использовать для сортировки, по умолчанию), search (использовать для поиска, когда важно только равенство или неравенство)

8.6. Другие методы класса String, чувствительные к локали

В классе String есть несколько методов, работающих с локалями. Метод localeCompare уже был описан в предыдущем разделе. Методы toLocaleUpperCase и toLocaleLowerCase принимают во внимание правила языка при изменении регистра. Например, в немецком языке буква «эс цет» ß в верхнем регистре записывается как две S:

```
'Großhändler'.toLocaleUpperCase('de') // 'GROSSHÄNDLER'
```

Метод localeCompare принимает те же параметры, что конструктор Intl.Collator из предыдущего раздела. Например, сравнение

```
'part10'.localeCompare('part2', 'en', { numeric: true })
```

возвращает положительное число, потому что при числовом сравнении строка 'part10' следует за 'part2'.

Иногда символ или последовательность символов можно выразить в Юникоде несколькими способами. Например, Å(\u{00C5}) можно также запи-

сать как простое A (\u{0041}), за которым следует комбинируемый кружок (\u{030A}).

Перед сохранением строки или передачей ее в другую программу желательно преобразовать строку в нормализованную форму. В стандарте Юникода определено четыре формы нормализации (C, D, KC, KD) – см. www.unicode.org/unicode/reports/tr15/tr15-23.html. В форме нормализации C символы с диакритическими знаками всегда составные. Например, последовательность A и ˚ комбинируется в один символ Å. В форме D символы с диакритическими знаками всегда разложены на базовую букву и комбинируемые диакритические знаки: Å преобразуется в A, за которой следует ˚. В формах KC и KD разлагаются такие символы, как знак торговой марки ™ (\u{2122}). Консорциум W3C рекомендует использовать форму нормализации C для передачи через интернет.

Весь этот процесс реализуется методом `normalize` класса `String`. Проверим его во всех четырех режимах. Для каждого режима показан результат вызова `normalize`, чтобы были хорошо видны отдельные символы.

```
const str = 'E™'
['NFC', 'NFD', 'NFKC', 'NFKD'].map(mode => [...str.normalize(mode)])
  // Возвращает ['Å', '™'], ['A', '˚', '™'], ['Å', 'T', 'M'], ['A', '˚', 'T', 'M']
```

8.7. Правила образования множественного числа и списков

Во многих языках имеются специальные формы существительных для малых величин. В английском счет ведется так: 0 dollars, 1 dollar, 2 dollars, 3 dollars и т. д. Форма слова dollar для количества 1 отличается от прочих.

С русскими рублями дело обстоит сложнее. Для 1 и «небольших» количеств есть специальные формы: 0 рублей, 1 рубль, 2, 3, 4 рубля. Начиная с 5 словоизменение восстанавливается: 5 рублей и т. д.

О таких правилах нужно помнить при форматировании сообщений типа «Найдено *n* соответствий».

Класс `Intl.PluralRules` помогает справиться с этой проблемой. Метод `select` возвращает ключ, который описывает, какая форма необходима для данного количества. Ниже приведены его результаты для английского и русского языков:

```
[0, 1, 2, 3, 4, 5].map(i => (new Intl.PluralRules('en').select(i)))
  // ['other', 'one', 'other', 'other', 'other', 'other']
[0, 1, 2, 3, 4, 5].map(i => (new Intl.PluralRules('ru').select(i)))
  // ['many', 'one', 'few', 'few', 'few', 'many']
```

Класс `PluralRules` порождает лишь английские названия форм. Их еще предстоит отобразить на локализованные словоформы. Для этого понадобится отображение для каждого языка:

```
dollars = { one: 'dollar', other: 'dollars' }
rubles = { one: 'рубль', few: 'рубля', many: 'рублей' }
```

После этого можно вызвать

```
dollars[new Intl.PluralRules('en').select(i)]
rubles[new Intl.PluralRules('ru').select(i)]
```

У метода `select` есть один дополнительный параметр:

- `type`: `cardinal` (по умолчанию), `ordinal`.

Посмотрим, что он выдает для английских порядковых числительных:

```
const rules = new Intl.PluralRules('en', { type: 'ordinal' })
[0, 1, 2, 3, 4, 5].map(i => rules.select(i))
  // ['other', 'one', 'two', 'few', 'other', 'other']
```

Что тут происходит? Оказывается, что английский язык не проще русского – порядковые числительные в нем имеют такие суффиксы: 0th, 1st, 2nd, 3rd, 4th, 5th и т. д.

Класс `Intl.ListFormat` помогает при форматировании списков значений. Понять его проще всего на примере:

```
let list = ['Goethe', 'Schiller', 'Lessing']
new Intl.ListFormat('en', { type: 'conjunction' }).format(list)
  // Возвращает строку 'Goethe, Schiller, and Lessing'
```

Как видим, метод `format` знает о соединительном союзе «and» и запятой перед ним.

Если параметр `type` равен `'disjunction'`, то элементы соединяются союзом «or». Посмотрим, что получается для немецкого языка:

```
new Intl.ListFormat('de', { type: 'disjunction' }).format(list)
  // 'Goethe, Schiller oder Lessing'
```

Метод `format` принимает следующие параметры:

- `type`: `conjunction` (по умолчанию), `disjunction`, `unit`;
- `style`: `long` (по умолчанию), `short`, `narrow` (только совместно с типом `unit`).

Тип `unit` предназначен для перечисления единиц измерения, например «7 pounds 11 ounces». К сожалению, если задан стиль `'long'` или `'short'`, то объект форматирования для английского языка порождает запятые:

```
list = ['7 pounds', '11 ounces']
new Intl.ListFormat('en', { type: 'unit', style: 'long' }).format(list)
  // '7 pounds, 11 ounces'
```

Ни чикагский стилистический справочник[1], ни руководство по стилю издательства Associated Press такого не одобрили бы.

---

[1] Одно из наиболее широко используемых и уважаемых руководств по стилю в США. – *Прим. перев.*

8.8. Различные средства, относящиеся к локалям

В современных браузерах свойство `navigator.languages` представляет собой массив тегов локалей в порядке убывания предпочтения пользователем. Свойство `navigator.language` содержит наиболее предпочтительный тег локали и совпадает с `navigator.languages[0]`. Браузеры обычно используют локаль операционной системы, если только пользователь не изменил языковые настройки.

Вы можете передавать `navigator.languages` в качестве локали различным чувствительным к локали методам и конструкторам, описанным в предыдущих разделах.

Метод `Intl.getCanonicalLocales` принимает тег локали или массив тегов локалей и возвращает массив очищенных тегов, из которого устранены дубликаты.

Во всех классах форматирования, описанных выше, имеется метод `supportedLocalesOf`. Если передать ему тег локали или массив таких тегов, то неподдерживаемые теги будут исключены, а поддерживаемые нормализованы. Например, в предположении, что класс `Intl.NumberFormat` в вашем браузере не поддерживает валлийский язык, вызов `Intl.NumberFormat.supportedLocalesOf(['cy', 'en-uk'])` вернет `['en-UK']`.

Если чувствительному к локали методу передан массив тегов локалей, то браузер сам находит локаль, лучше всего соответствующую предпочтениям. Все зависящие от локали функции поддерживают параметр `localeMatcher` для задания алгоритма сопоставления. Этот параметр может принимать два значения:

- `'lookup'` – использовать стандартный алгоритм, описанный в ECMA-402;
- `'best fit'` (по умолчанию) – разрешить исполняющей среде JavaScript поиск лучшего соответствия.

В настоящее время наиболее распространенные исполняющие среды JavaScript используют стандартный алгоритм, так что этот параметр можно не задавать.

> **Примечание.** Если вы хотите отдать выбор локали в руки пользователя, то должны будете отобразить список вариантов на языке, который пользователь понимает. На третьей стадии рассмотрения находится предложение, в котором определяется предназначенный для этой цели класс `Intl.DisplayNames`. Приведем несколько примеров его использования:

```
const regionNames = new Intl.DisplayNames(['fr'], { type: 'region' })
const languageNames = new Intl.DisplayNames(['fr'], { type: 'language' })
const currencyNames = new Intl.DisplayNames(['zh-Hans'],
  { type: 'currency' })
regionNames.of('US') // 'États-Unis'
languageNames.of('fr') // 'Français'
currencyNames.of('USD') // '世界'
```

Для получения дополнительной информации о свойствах объекта интернационализации служит метод `resolvedOptions`. Например, если имеется следующий объект сравнения

```
const collator = new Intl.Collator('en-US-u-kn-true', { sensitivity: 'base' })
```

то вызов

```
collator.resolvedOptions()
```

возвращает объект

```
{
  locale: 'en-US',
  usage: 'sort',
  sensitivity: 'base',
  ignorePunctuation: false,
  numeric: true,
  caseFirst: 'false',
  collation: 'default'
}
```

> **Примечание.** На стадии 3 рассмотрения находится класс `Intl.Locale`, предлагающий удобный способ обозначения локали с параметрами:
>
> ```
> const germanCurrency = new Intl.Locale('de-DE',
>   { style: 'currency', currency: 'EUR' })
> ```

Упражнения

1. Реализуйте класс `Person` с полями для имени, фамилии, пола и семейного положения. Включите метод `toLocaleString` для форматирования имени, например `'Ms. Smith'`, `'Frau Smith'`, `'Mme Smith'`. Узнайте, какие формы вежливости приняты в нескольких языках, и спроектируйте параметры для таких вариантов, как Ms. или Mrs./Miss.
2. Напишите программу, которая форматирует значение в виде числа, процента или суммы в долларах. Изучите все варианты отображения денежных величин. Попробуйте включить и выключить разделение на группы и разберитесь в смысле различных ограничений на количество цифр.
3. Посмотрите, как выглядят числа при использовании английских, арабских и тайских числительных. Какие еще варианты написания вы сумеете получить?
4. Напишите программу для демонстрации стилей форматирования даты и времени во Франции, Китае, Египте и Таиланде (с использованием тайских цифр).
5. Создайте массив, содержащий все двухбуквенные коды языков (по стандарту ISO 639-1). Для каждого кода отформатируйте дату и время. Сколько получилось разных форматов?

6. Напишите программу, которая выводит все символы Юникода, которые распадаются на два или более символов ASCII в формах нормализации KC или KD.
7. Приведите примеры, демонстрирующие различные параметры чувствительности при сравнении.
8. Что происходит в турецкой локали при переводе буквы `'i'` в верхний регистр или буквы `'I'` в нижний? Допустим, вы пишете программу, которая проверяет наличие HTTP-заголовка `If-Modified-Since`. В HTTP-заголовках регистр не имеет значения. Как организовать поиск заголовка, чтобы программа работала везде, в т. ч. в Турции?
9. В библиотеке Java имеется полезное понятие «пакета сообщений» (message bundle), которое позволяет искать локализованные сообщения по локали с предоставлением резервного варианта на случай отсутствия искомого. Предложите аналогичный механизм для JavaScript. Для каждой локали существует отображение ключей на переводы сообщений:

   ```
   { de: { greeting: 'Hallo', farewell: 'Auf Wiedersehen' },
     'de-CH' : { greeting: 'Grüezi' },
     fr: { greeting: 'Bonjour', farewell: 'Au revoir' },
     ...
   }
   ```

   При поиске сообщения сначала нужно просматривать самую специфичную локаль, а затем переходить к более общим. Организуйте поддержку более специализированных локалей. Например, при поиске сообщения с ключом `'greeting'` в локали `'de-CH'` возвращайте `'Grüezi'`, но для сообщения с ключом `'farewell'` остановитесь на локали `'de'`.
10. В библиотеке Java имеется полезный класс для форматирования зависящих от локали сообщений. Рассмотрим шаблон `'{0} has {1} messages'`. Его французская версия должна иметь вид `'Il y a {1} messages pour {0}'`. При форматировании сообщения аргументы передаются в фиксированном порядке, не зависящем от языка. Напишите функцию `messageFormat`, которая принимает шаблонную строку и переменное число аргументов. Придумайте механизм для включения самих фигурных скобок.
11. Предложите класс для зависящего от локали отображения размеров листа бумаги с использованием предпочтительной единицы измерения и размера по умолчанию для данной локали. Во всех странах, кроме США и Канады, размеры листов бумаги определяются стандартом ISO 216. Лишь три страны еще не перешли официально на метрическую систему мер: Либерия, Мьянма (Бирма) и США.

Глава 9

Асинхронное программирование

В этом разделе вы узнаете, как координировать задачи, которые могут быть выполнены в какой-то момент в будущем. Для начала мы подробно изучим понятие *обещания*. Обещание, как явствует из названия, – это действие, которое даст результат в будущем, если только этому не помешает исключение. Мы увидим, что обещания можно выполнять последовательно или параллельно.

У обещаний есть один недостаток – для их комбинирования нужно вызывать методы. Синтаксическая конструкция `async/await` намного приятнее. Мы пишем код с обычным потоком управления, а компилятор преобразует его в цепочку обещаний.

Вообще-то, можно было бы пропустить описание обещаний и сразу перейти к синтаксису `async/await`. Но мне кажется, что понять сложности и ограничения этой конструкции, не зная, что за ней стоит, было бы затруднительно.

Мы закончим эту главу обсуждением асинхронных генераторов и итераторов. Все разделы, кроме последнего, обязательны для прочтения JavaScript-разработчиками среднего уровня, поскольку асинхронная обработка встречается в веб-приложениях повсеместно.

9.1. Конкурентные задачи в JavaScript

Программа называется «конкурентной», если она выполняет различные действия в пересекающиеся отрезки времени. В Java и C++ в конкурентных программах используется несколько потоков выполнения. Если процессор многоядерный, то эти потоки действительно работают параллельно. Но имеется проблема – программист должен аккуратно защищать данные, чтобы избежать их порчи в результате одновременного обновления со стороны нескольких потоков.

С другой стороны, JavaScript-программа работает в одном-единственном потоке. В частности, если функция начала исполняться, то она доработает до конца, и только потом управление может быть передано другой части про-

граммы. Это хорошо. Мы точно знаем, что никакой код не сможет испортить данные, с которыми работает наша функция. В теле функции можно сколько угодно модифицировать переменные программы, нужно только не забыть прибраться перед выходом. Ни о каких мьютексах и взаимоблокировках можно не беспокоиться.

Проблема, связанная с наличием единственного потока, очевидна: если программе нужно дождаться какого-то события – чаще всего поступления данных из интернета, – то ничего другого она делать не может. Поэтому длительные операции в JavaScript всегда *асинхронны*. Мы говорим, что хотим сделать, и задаем функции обратного вызова, которые будут вызваны, когда поступят данные или произойдет ошибка. А текущая функция продолжает работать и может заниматься другими делами.

Рассмотрим простой пример – загрузку изображения. Приведенная ниже функция загружает изображение с заданным URL-адресом и добавляет его в конец заданного элемента DOM:

```
const addImage = (url, element) => {
  const request = new XMLHttpRequest()
  request.open('GET', url)
  request.responseType = 'blob'

  request.addEventListener('load', () => {
    if (request.status == 200) {
      const blob = new Blob([request.response], { type: 'image/png' })
      const img = document.createElement('img')
      img.src = URL.createObjectURL(blob)
      element.appendChild(img)
    } else {
      console.log(`${request.status}: ${request.statusText}`)
    }
  })
  request.addEventListener('error', event => console.log('Network error'));
  request.send()
}
```

Детали вызова функции `XMLHttpRequest` не так важны, кроме одного важного момента. Данные изображения обрабатываются внутри функции обратного вызова – прослушивателя события `load`.

Вызов `addImage` возвращает управление немедленно. Но изображение добавляется в элемент DOM гораздо позже, когда данные загрузятся.

Рассмотрим, что происходит при загрузке четырех изображений (взятых из колоды японских карт Ханафуда, см. `https://en.wikipedia.org/wiki/Hanafuda`):

```
const imgdiv = document.getElementById('images')
addImage('hanafuda/1-1.png', imgdiv)
addImage('hanafuda/1-2.png', imgdiv)
addImage('hanafuda/1-3.png', imgdiv)
addImage('hanafuda/1-4.png', imgdiv)
```

Все четыре вызова `addImage` возвращают управление немедленно. Как только приходят данные какого-то изображения, вызывается функция обратного вызова, и это изображение добавляется. Заметим, что беспокоиться о порче данных в результате конкурентных обратных вызовов нам не нужно. Обратные вызовы никогда не чередуются, а исполняются один за другим в одном потоке JavaScript. Однако запускаться они могут в любом порядке. Если несколько раз выполнить эту страницу, то порядок изображений может измениться (см. рис. 9.1).

> **Примечание.** Все примеры программ в данной главе предназначены для работы в браузере. В сопроводительном коде имеются страницы, которые можно загрузить в браузер, и фрагменты кода, которые можно вставить в консоль разработки. Для экспериментов с этими файлами в локальной файловой системе необходимо запустить локальный веб-сервер. Для установки `light-server` выполните команду
>
> `npm install -g light-server`
>
> Перейдите в каталог, содержащий отдаваемые файлы, и выполните команду
>
> `light-server -s .`
>
> После этого введите в адресной строке браузера URL-адрес `http://localhost:4000/images.html`.

Рис. 9.1 ❖ Изображения могут загружаться не по порядку

Справиться с приходом файлов не по порядку довольно просто – см. упражнение 1. Но рассмотрим более сложную ситуацию. Пусть требуется прочитать удаленные данные и в зависимости от того, что именно получено, прочитать дополнительные данные. Например, веб-страница может содержать URL-адрес изображения, которое тоже требуется загрузить.

В этом случае нужно асинхронно прочитать веб-страницу с помощью функции обратного вызова, которая ищет в содержимом URL-адрес изображения. Затем это изображение нужно асинхронно загрузить с помощью другой функции обратного вызова, которая помещает изображение в нужное место. При каждой загрузке возможна ошибка, для обработки которой необходимы дополнительные обратные вызовы. Достаточно всего нескольких уровней обработки, чтобы программа превратилась в «ад обратных вызо-

вов» – глубоко вложенные обратные вызовы, в которых трудно разобраться: что происходит в случае успеха, а что в случае ошибки.

В следующих разделах мы узнаем, как *обещания* помогают компоновать асинхронные задачи без вложенных обратных вызовов.

Обещание – это объект, который обещает произвести результат когда-нибудь в будущем, если получится. Быть может, результат станет доступен немедленно, а быть может, не будет получен никогда – если случится ошибка.

Звучит не слишком обнадеживающе, но, как мы скоро увидим, сцеплять действия, совершаемые в случае успеха и ошибки, с помощью обещаний куда проще, чем с помощью обратных вызовов.

9.2. Создание обещаний

В этом и следующем разделах мы увидим, как создавать обещания. Это технический вопрос, и вряд ли вам когда-нибудь придется делать подобное самостоятельно. Гораздо чаще вызывают библиотечную функцию, которая возвращает обещание. Можете спокойно пропустить данные разделы, пока не настанет момент, когда нужно будет сконструировать обещание вручную.

> **Примечание.** Типичный пример API, порождающего обещания, – *API выборки* (Fetch API), поддерживаемый всеми современными браузерами. Вызов
>
> `fetch('https://horstmann.com/javascript-impatient/hanafuda/index.html')`
>
> возвращает обещание, которое отдает ответ на HTTP-запрос, когда он становится доступным.

Конструктор объекта `Promise` принимает единственный аргумент – функцию с двумя аргументами: обработчиками успеха и ошибки. Эта функция называется «исполнителем».

```
const myPromise = new Promise((resolve, reject) => {
  // тело функции-исполнителя
})
```

В теле функции-исполнителя мы запускаем задачу, которая возвращает желаемый результат. Когда результат станет доступен, мы передаем его обработчику `resolve`. Или если известно, что никакого результата не будет, то вызываем обработчик `reject`, передавая ему причину ошибки. После асинхронного завершения работы эти обработчики будут вызваны в какой-то функции обратного вызова.

Вот схематичное описание процесса:

```
const myPromise = new Promise((resolve, reject) => {
  const callback = (args) => {
    ...
    if (успех) resolve(результат) else reject(причина)
  }
  invokeTask(callback)
})
```

Проделаем все это в очень простом случае: доставка результата после задержки. Следующая функция возвращает обещание сделать это:

```
const produceAfterDelay = (result, delay) => {
    return new Promise((resolve, reject) => {
      const callback = () => resolve(result)
      setTimeout(callback, delay)
    })
}
```

В функции-исполнителе, которая передается конструктору, мы вызываем функцию `setTimeout` с двумя аргументами: функция обратного вызова и величина задержки. Функция обратного вызова будет вызвана, когда истечет время задержки. Она передает результат обработчику `resolve`. Об ошибках тут можно не думать, поэтому обработчик `reject` не используется.

А вот пример более сложной функции, которая возвращает обещание вернуть изображение:

```
const loadImage = url => {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest()
    const callback = () => {
      if (request.status == 200) {
        const blob = new Blob([request.response], { type: 'image/png' })
        const img = document.createElement('img')
        img.src = URL.createObjectURL(blob)
        resolve(img)
      } else {
        reject(Error(`${request.status}: ${request.statusText}`))
      }
    }
    request.open('GET', url)
    request.responseType = 'blob'
    request.addEventListener('load', callback)
    request.addEventListener('error', event => reject(Error('Network error')));
    request.send()
  })
}
```

Функция-исполнитель конфигурирует объект `XMLHttpRequest` и отправляет его. После получения ответа обратный вызов порождает изображение и передает его обработчику `resolve`. Если же произойдет ошибка, то она будет передана обработчику `reject`.

Рассмотрим поток управления при работе с обещанием в режиме «замедленной съемки».

1. Вызывается конструктор `Promise`.
2. Вызывается функция-исполнитель.
3. Исполнитель инициирует асинхронную задачу с одним или несколькими обратными вызовами.
4. Исполнитель возвращает управление.
5. Конструктор возвращает управление. Теперь обещание находится в состоянии *ожидает* (pending).

6. Код, из которого вызван конструктор, дорабатывает до конца.
7. Асинхронная задача завершается.
8. Вызывается функция обратного вызова, ассоциированная с задачей.
9. Эта функция вызывает обработчик `resolve` или `reject`, и обещание переходит в состояние *выполнено* (fulfilled) или *отвергнуто* (rejected). В любом случае обещание *улажено*.

> **Примечание.** На последнем шаге этого потока управления возможен другой поворот событий. Мы можем вызвать `resolve`, передав ему еще одно обещание. Тогда текущее обещание считается разрешенным, но еще не выполненным. Оно остается в состоянии *ожидает*, пока не будет улажено следующее обещание. Именно по этой причине функция-обработчик называется `resolve`, а не `fulfill`.

Не забывайте вызывать `resolve` или `reject` в своих обратных вызовах, иначе обещание никогда не выйдет из состояния *ожидает*.

Это означает, что вы должны обращать внимание на исключения в обратных вызовах, ассоциированных с задачей. Если обратный вызов завершается вследствие исключения, а не вызова `resolve` или `reject`, то обещание не может быть улажено. В примере `loadImage` я тщательно написал код таким образом, чтобы он не возбуждал никаких исключений. В общем случае рекомендуется использовать в обратном вызове предложение `try/catch` и передавать все исключения обработчику `reject`.

Но если исключение возникает в функции-исполнителе, то перехватывать его не нужно. Конструктор просто вернет обещание в состояние *отвергнуто*.

9.3. Немедленно улаживаемые обещания

Вызов `Promise.resolve(value)` создает обещание, которое немедленно выполняется, возвращая заданное значение. Это полезно в методах, которые возвращают обещания, и в некоторых случаях, когда ответ известен сразу:

```
const loadImage = url => {
  if (url === undefined) return Promise.resolve(brokenImage)
    ...
}
```

Если `value` может быть обещанием или простым значением, то `Promise.resolve(value)` гарантированно преобразует его в обещание. Если значение уже является обещанием, то оно просто возвращается без изменения.

> **Примечание.** Для совместимости с библиотеками, написанными до принятия стандарта обещаний ECMAScript, метод `Promise.resolve` ведет себя специальным образом, если `value` является объектом, имеющим метод `then`. Этому методу передаются обработчики `resolve` и `reject`, а возвращенное обещание будет находиться в том состоянии, в которое его перевел объект `value` (см. упражнение 6).

Вызов `Promise.reject(error)` возвращает обещание, которое немедленно отвергается с заданной ошибкой. Используйте его, когда функция, порождающая обещание, завершается с ошибкой:

```
const loadImage = url => {
  if (url === undefined) {
    return Promise.reject(Error('No URL'))
  } else {
    return new Promise(...)
  }
}
```

9.4. Получение результата обещания

Итак, мы знаем, как сконструировать обещание, а теперь хотим получить его результат. Мы не ждем, когда обещание будет улажено, а вместо этого задаем действия, которые обрабатывают результат или ошибку, как только это произойдет. Эти действия выполняются в какой-то момент *после* завершения запланировавшей их функции.

Используйте метод `then` для задания действия, которое следует выполнить после разрешения обещания. Действием является функция, которой передается результат.

```
const promise1 = produceAfterDelay(42, 1000)
promise1.then(console.log) // протоколировать значение по готовности

const promise2 = loadImage('hanafuda/1-1.png')
promise2.then(img => imgdiv.appendChild(img)) // добавить изображение, когда
                                              // оно будет получено
```

Примечание. Метод `then` – единственный способ получить результат от обещания.

В разделе 9.6 мы увидим, как следует поступать с отвергнутыми обещаниями.

Предостережение. В ходе экспериментов с функциями `loadImage` и `fetch`, вызываемыми с разными URL-адресами, легко наткнуться на ошибку «cross-origin» (другой источник). Движок JavaScript внутри браузера не позволяет JavaScript-коду видеть результаты веб-запросов от сторонних серверов, если только серверы не договорились о безопасности доступа и не установили специальный заголовок в ответе. К сожалению, немногие сайты готовы пройти все эти процедуры. Вы можете беспрепятственно обратиться к URL-адресам https://horstmann.com/javascript-impatient или (по состоянию на момент написания книги) https://developer.mozilla.org и https://aws.random.cat/meow. Если хотите поэкспериментировать с другими сайтами, то можете воспользоваться CORS-прокси или подключаемым к браузеру модулем для обхода выполняемой браузером проверки.

9.5. Сцепление обещаний

В предыдущем разделе мы видели, как получить результат обещания. А теперь рассмотрим более интересный случай, когда результат обещания передается другой асинхронной задаче.

Если действие, которое передано методу then, возвращает другое обещание, то результатом является это обещание. Чтобы обработать его результат, нужно еще раз вызвать метод then.

Приведем пример – мы загружаем одно изображение, а затем другое:

```
const promise1 = loadImage('hanafuda/1-1.png')
const promise2 = promise1.then(img => {
  imgdiv.appendChild(img)
  return loadImage('hanafuda/1-2.png') // еще одно обещание
})
promise2.then(img => {
  imgdiv.appendChild(img)
})
```

Необязательно сохранять каждое обещание в отдельной переменной. Обычно цепочка обещаний обрабатывается «конвейерным» способом:

```
loadImage('hanafuda/1-1.png')
  .then(img => {
    imgdiv.appendChild(img)
    return loadImage('hanafuda/1-2.png')
  })
  .then(img => imgdiv.appendChild(img))
```

> **Примечание.** При использовании API выборки обещания необходимо сцеплять для чтения содержимого веб-страницы:
>
> ```
> fetch('https://developer.mozilla.org')
>   .then(response => response.text())
>   .then(console.log)
> ```
>
> Функция fetch возвращает обещание, которое отдает ответ, а метод text возвращает еще одно обещание для чтения текстового содержимого страницы.

Синхронные и асинхронные задачи можно перемешивать:

```
loadImage('hanafuda/1-1.png')
  .then(img => imgdiv.appendChild(img)) // синхронно
  .then(() => loadImage('hanafuda/1-2.png')) // асинхронно
  .then(img => imgdiv.appendChild(img)) // синхронно
```

Технически, если действие then возвращает значение, не являющееся обещанием, метод then возвращает немедленно выполненное обещание. Это позволяет затем произвести сцепление с другим методом then.

> **Совет.** Конвейеры обещаний можно сделать более симметричными, начав с немедленно выполненного обещания:
>
> ```
> Promise.resolve()
>   .then(() => loadImage('hanafuda/1-1.png'))
>   .then(img => imgdiv.appendChild(img))
>   .then(() => loadImage('hanafuda/1-2.png'))
>   .then(img => imgdiv.appendChild(img))
> ```

В предыдущих примерах показано, как компонуется фиксированное количество задач. Можно построить произвольно длинный конвейер в цикле:

```
let p = Promise.resolve()
for (let i = 1; i <= n; i++) {
  p = p.then(() => loadImage(`hanafuda/1-${i}.png`))
    .then(img => imgdiv.appendChild(img))
}
```

Предостережение. Если аргумент метода then – не функция, то он отбрасывается! Следующий код некорректен:

```
loadImage('hanafuda/1-1.png')
  .then(img => imgdiv.appendChild(img))
  .then(loadImage('hanafuda/1-2.png'))
    // Ошибка - аргумент then не является функцией
  .then(img => imgdiv.appendChild(img))
```

Здесь методу then передано значение, возвращенное функцией loadImage, т. е. объект Promise. Если вызвать p.then(arg) с отличным от функции аргументом, то никакого сообщения об ошибке не будет. Аргумент просто игнорируется, и метод then возвращает обещание с таким же результатом, как p. Кроме того, отметим, что второе обращение к loadImage производится сразу после первого, не дожидаясь улаживания первого обещания.

9.6. Обработка отвергнутых обещаний

В предыдущем разделе мы видели, как организовать последовательность нескольких асинхронных задач. Мы рассматривали только благоприятный сценарий, когда все задачи успешно завершаются. Обработка ошибок может существенно усложнить логику программы. Обещания позволяют сравнительно просто распространять ошибки по конвейеру задач.

При вызове метода then можно указать обработчик ошибок:

```
loadImage(url)
  .then(
    img => { // обещание улажено
      imgdiv.appendChild(img)
    },
    reason => { // обещание было отвергнуто
      console.log({reason})
      imgdiv.appendChild(brokenImage)
})
```

Однако обычно лучше использовать метод catch:

```
loadImage(url)
  .then(
    img => { // обещание улажено
      imgdiv.appendChild(img)
    })
  .catch(
    reason => { // предыдущее обещание было отвергнуто
      console.log({reason})
```

```
    imgdiv.appendChild(brokenImage)
  })
```

В этом случае перехватываются также ошибки в обработчике `resolve`.

Метод `catch` возвращает новое обещание, основанное на возвращенном значении, возвращенном обещании или исключении, возбужденном обработчиком, который был указан в аргументе `handler`.

Если обработчик возвращает управление не вследствие исключения, то результирующее обещание улажено и конвейер может продолжить работу.

Часто в конвейере имеется единственный обработчик отвергнутых обещаний, который вызывается в случае ошибки в любой задаче:

```
Promise.resolve()
  .then(() => loadImage('hanafuda/1-1.png'))
  .then(img => imgdiv.appendChild(img))
  .then(() => loadImage('hanafuda/1-2.png'))
  .then(img => imgdiv.appendChild(img))
  .catch(reason => console.log({reason}))
```

Если в методе `then` возникло исключение, то он возвращает отвергнутое обещание. Сцепление отвергнутого обещания с другим методом `then` приводит к распространению этого обещания дальше. Поэтому обработчик, переданный методу `catch`, находящемуся в самом конце, сможет обработать отвержение на любой стадии конвейера.

Метод `finally` вызывает обработчик вне зависимости от того, было улажено обещание или нет. У этого обработчика нет аргументов, потому что он предназначен для очистки, а не для анализа результата обещания. Метод `finally` возвращает обещание с тем же исходом, что и то, для которого он вызван, поэтому его можно включить в конвейер:

```
Promise.resolve()
  .then(() => loadImage('hanafuda/1-1.png'))
  .then(img => imgdiv.appendChild(img))
  .finally(() => { doCleanup(...) })
  .catch(reason => console.log({reason}))
```

9.7. Выполнение нескольких обещаний

Если необходимо разрешить несколько обещаний, то можно поместить их в массив или в любой итерируемый объект и вызвать метод `Promise.all(iterable)`. Затем можно получить обещание, которое будет разрешенным, если разрешены все обещания в итерируемом объекте. Значением такого комбинированного обещания является итерируемый объект, содержащий результаты всех обещаний в том же порядке, в каком расположены сами обещания.

Это дает простой способ загрузить последовательность изображений и добавить их в нужном порядке:

```
const promises = [
  loadImage('hanafuda/1-1.png'),
  loadImage('hanafuda/1-2.png'),
  loadImage('hanafuda/1-3.png'),
  loadImage('hanafuda/1-4.png')]
Promise.all(promises)
  .then(images => { for (const img of images) imgdiv.appendChild(img) })
```

Метод `Promise.all` не организует параллельное выполнение задач. Все задачи выполняются последовательно в одном потоке. Однако порядок их планирования непредсказуем. Например, в примере загрузки сообщений мы не знаем, какое изображение придет первым.

Как уже отмечалось, `Promise.all` возвращает обещание, содержащее итерируемый объект в качестве значения. В этом объекте хранятся результаты отдельных обещаний в правильном порядке вне зависимости от того, в каком порядке они были получены.

В предыдущем примере метод `then` вызывается после загрузки всех изображений, и они добавляются из итерируемого объекта `images` в правильном порядке.

Если итерируемый объект, переданный методу `Promise.all`, содержит значения, отличные от обещаний, то они просто включаются в результирующий итерируемый объект.

Если какое-то обещание отвергается, то `Promise.all` возвращает отвергнутое обещание, в котором хранится ошибка из первого отвергнутого обещания.

Если требуется более детальный контроль над отвергнутыми обещаниями, воспользуйтесь методом `Promise.allSettled`. Он возвращает обещание, значением которого является итерируемый объект с элементами вида

```
{ status: 'fulfilled', value: результат }
```

или

```
{ status: 'rejected', reason: исключение }
```

В упражнении 8 показано, как обрабатывать такие результаты.

9.8. Гонка нескольких обещаний

Иногда требуется выполнить несколько задач параллельно, но остановиться, как только первая задача завершится. Типичный пример – поиск, когда нам достаточно первого результата. Метод `Promise.race(iterable)` выполняет обещания, хранящиеся в итерируемом объекте, до тех пор, пока какое-нибудь не будет улажено. Это обещание и определяет исход гонки.

> **Предостережение.** Если в итерируемом объекте имеются значения, отличные от обещаний, то одно из них станет результатом гонки. Если итерируемый объект пуст, то обещание `Promise.race(iterable)` никогда не будет улажено.

Может случиться так, что гонку выиграет отвергнутое обещание. В таком случае все остальные обещания отбрасываются, даже если одно из них могло бы дать успешный результат. Более полезный метод, `Promise.any`, в настоящее время находится на третьей стадии рассмотрения.

Метод `Promise.any` продолжает работу до тех пор, пока какое-нибудь обещание не будет *разрешено*. В том печальном случае, когда все обещания отвергаются, результирующее обещание будет находиться в состоянии *отвергнуто*, а ошибка будет представлена объектом `AggregateError`, который содержит причины всех ошибок.

```
Promise.any(promises)
  .then(result => . . .) // обработать результат первого улаженного обещания
  .catch(error => . . .) // ни одно обещание не улажено
```

9.9. Асинхронные функции

Выше мы видели, как строить конвейеры из обещаний с помощью методов `then` и `catch` и как выполнять последовательность обещаний конкурентно с помощью методов `Promise.all` и `Promise.any`. Однако такой стиль программирования не очень удобен. Вместо использования знакомых последовательностей предложений и потока управления мы должны организовывать конвейер, вызывая методы.

Синтаксис `async/await` делает работу с обещаниями гораздо более естественной.

Выражение

```
let value = await promise
```

дожидается улаживания обещания и возвращает его значение.

Но секундочку... разве в начале этой главы мы не договорились, что ожидание в JavaScript-функции – никуда не годная идея? Да, так оно и есть, и использовать `await` в обычной функции нельзя. Оператор `await` может встречаться только в функции, помеченной ключевым словом `async`:

```
const putImage = async (url, element) => {
  const img = await loadImage(url)
  element.appendChild(img)
}
```

Компилятор преобразует код функции `async` таким образом, что любые действия, следующие за оператором `await`, выполняются, после того как обещание разрешится. Например, приведенная выше функция `putImage` эквивалентна такой:

```
const putImage = (url, element) => {
  loadImage(url)
    .then(img => element.appendChild(img))
}
```

Допускается несколько await:

```
const putTwoImages = async (url1, url2, element) => {
  const img1 = await loadImage(url1)
  element.appendChild(img1)
  const img2 = await loadImage(url2)
  element.appendChild(img2)
}
```

Циклы тоже разрешены:

```
const putImages = async (urls, element) => {
  for (url of urls) {
    const img = await loadImage(url)
    element.appendChild(img)
  }
}
```

Как видно из примеров, переписывание кода, осуществляемое компилятором за кулисами, далеко не тривиально.

> **Предостережение.** Если вы забудете поставить ключевое слово await при вызове асинхронной функции, то функция будет вызвана и вернет обещание, но это обещание никем не обрабатывается. Рассмотрим пример, взятый из одного блога:
>
> ```
> const putImages = async (urls, element) => {
>   for (url of urls)
>     putImage(url, element) // ошибка - нет await перед вызовом
>                            // асинхронной функции putImage
> }
> ```
>
> Эта функция порождает объекты Promise и забывает о них, а затем возвращает Promise.resolve(undefined). Если все пройдет хорошо, то изображения будут добавлены в каком-то порядке. Но если возникнет исключение, то его никто не перехватит.

Ключевое слово async можно применять к:

- стрелочным функциям:

  ```
  async url => { ... }
  async (url, params) => { ... }
  ```

- методам:

  ```
  class ImageLoader {
   async load(url) { ... }
  }
  ```

- именованным и анонимным функциям:

  ```
  async function loadImage(url) { ... }
  async function(url) { ... }
  ```

- методам объектных литералов:

```
obj = {
 async loadImage(url) { ... },
 ...
}
```

Примечание. В любом случае результирующая функция – экземпляр класса AsyncFunction, а не Function, хотя typeof по-прежнему сообщает, что это 'function'.

9.10. Асинхронно возвращаемые значения

Асинхронная функция выглядит так, будто вернула значение, но на самом деле она всегда возвращает обещание. Например, сайт https://aws.random.cat/meow возвращает адреса случайных изображений кошек как JSON-объект вида { file: 'https://purr.objects-us-east-1.dream.io/i/mDh7a.jpg' }.

С помощью API выборки мы можем получить обещание загрузки содержимого:

```
const result = await fetch('https://aws.random.cat/meow')
const imageJSON = await result.json()
```

Второй await необходим, потому что в API выборки обработка JSON асинхронна – вызов result.json() возвращает еще одно обещание.

Теперь мы готовы написать функцию, которая возвращает URL-адрес изображения кошки:

```
const getCatImageURL = async () => {
  const result = await fetch('https://aws.random.cat/meow')
  const imageJSON = await result.json()
  return imageJSON.file
}
```

Разумеется, функция должна быть помечена async, потому что в ней встречается оператор await.

Выглядит эта функция так, будто возвращает строку. Идея оператора await в том и заключается, чтобы мы могли работать со значениями, а не с обещаниями. Но эта иллюзия пропадает, как только мы покидаем асинхронную функцию. Значение в предложении return всегда становится обещанием.

Что можно сделать с помощью асинхронной функции? Поскольку она возвращает обещание, его результат можно получить, вызвав метод then:

```
getCatImageURL()
  .then(url => loadImage(url))
  .then(img => imgdiv.appendChild(img))
```

Или с помощью оператора `await`:

```
const url = await getCatImageURL()
const img = await loadImage(url)
imgdiv.appendChild(img)
```

Последний вариант симпатичнее, но он должен находиться еще в одной асинхронной функции. Как видим, оказавшись в асинхронном мире, выбраться из него трудно.

Рассмотрим последнюю строку следующей асинхронной функции:

```
const loadCatImage = async () => {
  const result = await fetch('https://aws.random.cat/meow')
  const imageJSON = await result.json()
  return await loadImage(imageJSON.file)
}
```

Последний оператор `await` можно опустить:

```
const loadCatImage = async () => {
  const result = await fetch('https://aws.random.cat/meow')
  const imageJSON = await result.json()
  return loadImage(imageJSON.file)
}
```

В любом случае эта функция возвращает обещание загрузить изображение, а это обещание было асинхронно порождено обращением к `loadImage`.

Я полагаю, что первый вариант понять проще, потому что синтаксис `async`/`await` упорно скрывает все обещания.

> **Предостережение.** Внутри предложения `try`/`catch` имеется тонкое различие между `return await promise` и `return promise` – см. упражнение 11. В данном случае не стоит опускать оператор `await`.

Если асинхронная функция возвращает значение, еще не дойдя до `await`, то это значение обертывается разрешенным обещанием:

```
const getJSONProperty = async (url, key) => {
  if (url === undefined) return null
    // на самом деле возвращается Promise.resolve(null)
  const result = await fetch(url)
  const json = await result.json()
  return json[key]
}
```

> **Примечание.** Асинхронные функции, описанные в этом разделе, возвращают одно значение в будущем. В главе 12 мы увидим, что асинхронный итератор порождает последовательность значений в будущем. Ниже приведен пример порождения диапазона целых чисел с заданной задержкой.
>
> ```
> async function* range(start, end, delay) {
>   for (let current = start; current < end; current++) {
>     yield await produceAfterDelay(current, delay)
>   }
> }
> ```

Пусть вас не пугает синтаксис этой «асинхронной генераторной функции». Вряд ли вам придется реализовывать нечто подобное, но можно воспользоваться той, что есть в библиотеке. Получить результаты можно в цикле `for await`:

```
for await (const value of range(0, 10, 1000)) {
  console.log(value)
}
```

Этот цикл должен находиться внутри асинхронной функции, поскольку он ожидает поступления всех значений.

9.11. Конкурентное ожидание

Последовательные обращения к `await` следуют один за другим:

```
const img1 = await loadImage(url)
const img2 = await loadCatImage() // начинается только после загрузки первого изображения
```

Было бы эффективнее загружать изображения одновременно. В таком случае нужно использовать метод `Promise.all`:

```
const [img1, img2] = await Promise.all([loadImage(url), loadCatImage()])
```

Чтобы разобраться, в чем смысл этого выражения, недостаточно понимать синтаксис `async/await`. Необходимо знать о самих обещаниях.

Аргументом метода `Promise.all` является итерируемый объект, содержащий обещания. Здесь `loadImage` – обычная функция, которая возвращает обещание, а `loadCatImage` – асинхронная функция, которая возвращает обещание неявно.

Метод `Promise.all` возвращает обещание, поэтому к нему можно применить `await`. Результатом обещания является массив, который мы деструктурируем.

Не понимая, что происходит под капотом, можно наделать ошибок. Рассмотрим такое предложение:

```
const [img1, img2] = Promise.all([await loadImage(url), await loadCatImage()])
  // ошибка - по-прежнему последовательно
```

Это предложение компилируется и выполняется. Но изображения не загружаются конкурентно. Вызов `await loadImage(url)` должен завершиться, прежде чем начнется `await loadCatImage()`.

9.12. Исключения в асинхронных функциях

Возбуждение исключения в асинхронной функции приводит к отвергнутому обещанию:

```
const getAnimalImageURL = async type => {
  if (type === 'cat') {
    return getJSONProperty('https://aws.random.cat/meow', 'file')
```

```
  } else if (type === 'dog') {
    return getJSONProperty('https://dog.ceo/api/breeds/image/random', 'message')
  } else {
    throw Error('bad type') // асинхронная функция возвращает отвергнутое обещание
  }
}
```

Обратно, когда оператор `await` получает отвергнутое обещание, он возбуждает исключение. Следующая функция перехватывает исключение от оператора `await`:

```
const getAnimalImage = async type => {
  try {
    const url = await getAnimalImageURL(type)
    return loadImage(url)
  } catch {
    return brokenImage
  }
}
```

Необязательно окружать каждый `await` предложением `try/catch`, но нужно иметь какую-то стратегию обработки ошибок в асинхронных функциях. Быть может, все такие исключения перехватываются в асинхронных функциях верхнего уровня, или же вы явно оговариваете в документации, что вызывающая сторона должна вызывать метод `catch` для возвращенного обещания.

В Node.js если обещание отвергается на верхнем уровне, то выдается суровое предупреждение о том, что в будущих версиях Node.js может в таком случае завершить процесс (см. упражнение 12).

Упражнения

1. Программа из раздела 9.1 может загружать изображения не в том порядке. Как модифицировать ее без использования обещаний, чтобы изображения всегда добавлялись в правильном порядке вне зависимости от того, в каком порядке они поступают?
2. Напишите функцию `invokeAfterDelay`, возвращающую обещание, которое вызывает заданную функцию с заданной задержкой. Продемонстрируйте ее работу, возвращая обещание, содержащее случайное число от 0 до 1. Полученный результат печатайте на консоли.
3. Вызовите функцию `produceRandomAfterDelay` два раза и напечатайте сумму, после того как будут получены оба слагаемых.
4. Напишите цикл, который вызывает функцию `produceRandomAfterDelay` из предыдущего упражнения `n` раз и печатает сумму, после того как будут получены все слагаемые.
5. Напишите функцию `addImage(url, element)`, похожую на приведенную в разделе 9.1. Верните обещание, так чтобы вызовы можно было сцеплять следующим образом:

```
addImage('hanafuda/1-1.png')
 .then(() => addImage('hanafuda/1-2.png', imgdiv))
 .then(() => addImage('hanafuda/1-3.png', imgdiv))
 .then(() => addImage('hanafuda/1-4.png', imgdiv))
```

Затем воспользуйтесь советом из раздела 9.5, чтобы сделать цепочку более симметричной.

6. Продемонстрируйте, что метод `Promise.resolve` преобразует любой объект, имеющий метод `then`, в объект `Promise`. Передайте ему объект, метод `then` которого случайным образом вызывает обработчик `resolve` или `reject`.
7. Часто бывает, что клиентскому приложению необходимо отложить некоторую работу до тех пор, пока браузер не закончит загрузку DOM. Такую работу можно поместить в обработчик события `DOMContentLoaded`. Но если `document.readyState != 'loading'`, то загрузка уже завершилась, и это событие больше не произойдет. Учтите оба случая в функции, возвращающей обещание, так чтобы ее можно было вызвать следующим образом: `whenDOMContentLoaded().then(...)`.
8. Создайте массив URL-адресов изображений, как хороших, так и не загружающихся из-за ограничения браузера CORS (см. примечание в конце раздела 9.2). Преобразуйте каждый адрес в обещание

```
const urls = [...]
const promises = urls.map(loadImage)
```

Вызовите метод `allSettled` для массива обещаний. Когда возвращенное им обещание разрешится, обойдите массив, добавьте загруженные изображения в элементы DOM и запротоколируйте те, что загрузить не удалось:

```
Promise.allSettled(promises)
 .then(results => {
   for (result of results)
     if (result.status === 'fulfilled') . . . else . . .
   })
```

9. Повторите предыдущее упражнение, воспользовавшись `await` вместо `then`.
10. Напишите функцию `sleep`, возвращающую обещание, чтобы можно было вызвать ее следующим образом:

```
await sleep(1000)
```

11. Опишите, в чем различие между

```
const loadCatImage = async () => {
 try {
   const result = await fetch('https://aws.random.cat/meow')
   const imageJSON = await result.json()
   return loadImage(imageJSON.file)
 } catch {
```

```
    return brokenImage
  }
}
```

и

```
const loadCatImage = async () => {
 try {
   const result = await fetch('https://aws.random.cat/meow')
   const imageJSON = await result.json()
   return await loadImage(imageJSON.file)
 } catch {
   return brokenImage
 }
}
```

Указание: что будет, если обещание, возвращенное функцией `loadImage`, отвергнуто?

12. Поэкспериментируйте с вызовом в Node.js асинхронной функции `async`, возбуждающей исключение. Пусть имеется функция

```
const rejectAfterDelay = (result, delay) => {
 return new Promise((resolve, reject) => {
   const callback = () => reject(result)
   setTimeout(callback, delay)
 })
}
```

Попробуйте выполнить предложение

```
const errorAfterDelay = async (message, delay) =>
 await rejectAfterDelay(new Error(message), delay)
```

Теперь вызовите функцию `errorAfterDelay`. Что произойдет? Как избежать такой ситуации?

13. Объясните, чем сообщение об ошибке из предыдущего упражнения может быть полезно для обнаружения забытого оператора `await`, например в таком фрагменте:

```
const errorAfterDelay = async (message, delay) => {
 try {
   return rejectAfterDelay(new Error(message), 1000)
 } catch(e) { console.error(e) }
}
```

14. Напишите полные программы для демонстрации функций `Promise.all` и `Promise.race` из раздела 9.7.
15. Напишите функцию `produceAfterRandomDelay`, которая порождает значение после случайной задержки (в миллисекундах) величиной от 0 до заданного числа. Затем создайте массив обещаний, полученных применением этой функции к числам 1, 2, ..., 10, и передайте его методу `Promise.all`. В каком порядке будут собраны результаты?

16. Используйте API выборки для загрузки изображения (не нарушая ограничение CORS). Загрузите данные из URL-адреса, затем вызовите для ответа метод `blob()`, чтобы получить обещание, содержащее BLOB. Преобразуйте его в изображение, как в функции `loadImage`. Предложите две реализации, с использованием `then` и `await`.
17. Используйте API выборки для получения HTML-кода веб-страницы (не нарушающей ограничение CORS). Найдите на этой странице все URL-адреса изображений и загрузите изображения.
18. Бывает так, что запланированная на будущее работа в силу изменившихся обстоятельств больше не нужна, и хорошо бы ее отменить. Спроектируем схему такой отмены. Рассмотрим многошаговый процесс, как в предыдущем упражнении. На каждом шаге должна быть возможность прервать процесс. Пока в JavaScript нет стандартного способа сделать это, но обычно API предоставляют «маркеры отмены». Функция `fetchImages` могла бы получать дополнительный аргумент:

    ```
    const token = new CancellationToken()
    const images = fetchImages(url, token)
    ```

    Вызывающая сторона впоследствии может вызвать метод

    ```
    token.cancel()
    ```

    В реализации допускающей отмену асинхронной функции вызов

    ```
    token.throwIfCancellationRequested()
    ```

    возбуждает исключение, если действительно была запрошена отмена. Реализуйте этот механизм и продемонстрируйте его работу на примере.
19. Рассмотрим следующий код, который выполняет какую-то асинхронную работу, например загружает удаленные данные, обрабатывает их и возвращает обещание дальнейшей обработки:

    ```
    const doAsyncWorkAndThen = handler => {
     const promise = asyncWork();
     promise.then(result => handler(result));
     return promise;
    }
    ```

    Что случится, если `handler` возбудит исключение? Как следует реорганизовать этот код?
20. Что случится, если добавить ключевое слово `async` к функции, которая не возвращает обещания?
21. Что случится, если применить оператор `await` к выражению, не являющемуся обещанием? А что будет, если это выражение возбудит исключение? Есть ли какая-нибудь причина желать, чтобы было именно так?

Глава 10

Модули

При написании кода, предназначенного для использования многими программистами, важно разделять открытый интерфейс и закрытую реализацию. В объектно-ориентированном программировании это разделение достигается с помощью классов. Если класс развивается путем изменения закрытой реализации, то на его пользователях это никак не отражается. (В главе 4 мы видели, что сокрытие возможностей еще не полностью поддерживается в JavaScript, но со временем это обязательно будет сделано.)

Система модулей предлагает те же преимущества, но в более крупном масштабе. Модуль может открыть доступ к одним классам и функциям и закрыть к другим, с тем чтобы эволюцию модуля можно было контролировать.

Для JavaScript было разработано несколько нестандартных систем модулей. Но в 2015 году в спецификации ECMAScript 6 была стандартизована простая система модулей, которая и является темой этой короткой главы.

10.1. Понятие модуля

Модуль предоставляет программистам средства (классы, функции и прочее), которые называются *экспортируемыми*. Все средства, кроме экспортированных, являются закрытым достоянием модуля.

В описании модуля также указано, от каких других модулей он зависит. Если нужен какой-то модуль, то исполняющая среда JavaScript загружает его и все модули, от которых он зависит.

Модуль предотвращает конфликты имен. Поскольку закрытые средства модуля скрыты от публики, не важно, как они называются. Они никогда не вступят в конфликт с именами вне модуля. Если используется открытое средство, то его можно переименовать, так чтобы имя было уникально.

> **Примечание.** В этом отношении модули JavaScript отличаются от пакетов или модулей Java, которые опираются на глобально уникальные имена.

Важно понимать, что модуль отличается от класса. У класса может быть много экземпляров, а у модуля экземпляров нет. Это просто контейнер для классов, функций или значений.

10.2. Модули в ECMAScript

Рассмотрим JavaScript-разработчика, который хочет сделать написанные им средства доступными другим программистам. Разработчик помещает эти средства в файл, а программист, которому они нужны, включает этот файл в свой проект.

Теперь предположим, что программист включает файлы, подготовленные несколькими разработчиками. Очень может быть, что имена некоторых средств будут конфликтовать друг с другом. Хуже того, в каждом файле есть вспомогательные функции и переменные, имена которых могут стать причиной дополнительных конфликтов.

Очевидно, нужен какой-то способ скрыть детали реализации. В течение многих лет JavaScript-разработчики имитировали модули с помощью замыканий, помещая вспомогательные функции и классы в функцию-обертку. Это напоминает технику «крепких объектов» из главы 3. Были также разработаны специализированные способы публикации экспортируемых средств и зависимостей.

В Node.js реализована система модулей (названная Common.js) для управления зависимостями. Если требуется некоторый модуль, то загружается он сам и все его зависимости. Эта загрузка производится *синхронно* в тот момент, когда возникает потребность в модуле.

Стандарт AMD (Asynchronous Module Definition – определение асинхронных модулей) определяет систему асинхронной загрузки модулей, которая лучше отвечает потребностям приложений, работающих в браузере.

Модули в ECMAScript являются усовершенствованием обеих систем. Они подвергаются синтаксическому разбору, который позволяет быстро выявить зависимости и экспортируемые элементы без предварительного выполнения тела модуля. Это открывает возможность асинхронной загрузки и не запрещает циклические зависимости. В настоящее время мир JavaScript постепенно переходит на систему модулей ECMAScript.

> **Примечание.** Для Java-программистов аналогом модуля JavaScript является артефакт Maven или, начиная с версии Java 9, платформенный модуль Java. Артефакты сообщают информацию о зависимостях, но не обеспечивают никакой инкапсуляции (помимо классов и пакетов Java). Платформенные модули Java дают то и другое, но они существенно сложнее, чем модули ECMAScript.

10.3. Импорт по умолчанию

Лишь немногие программисты пишут модули гораздо больше тех, кто их потребляет. Поэтому начнем с самой распространенной операции – импорта средств из существующего модуля.

Чаще всего импортируются функции и классы. Но можно импортировать также объекты, массивы и примитивные значения.

Автор модуля может пометить какое-то одно средство (надо думать, самое полезное) как подразумеваемое *по умолчанию*. Синтаксическая конструкция `import` позволяет особенно легко импортировать средство по умолчанию. В качестве примера рассмотрим импорт класса из модуля, предоставляющего средства шифрования:

```
import CaesarCipher from './modules/caesar.mjs'
```

Здесь указано имя, которое мы решили дать средству по умолчанию, а за ним следует файл, содержащий реализацию модуля. Подробнее о задании местоположения модулей см. раздел 10.7.

Решение о том, как называть средство в своей программе, остается за вами. Если хотите, можете выбрать более короткое имя:

```
import CC from './modules/caesar.mjs'
```

Если вы работаете с модулями, которые предоставляют свои возможности в виде средств по умолчанию, то это все, что нужно знать о системе модулей в ECMAScript.

> **Примечание.** В браузере местоположение модуля может быть задано абсолютным или относительным URL-адресом, начинающимся с `./`, `../` или `/`. Это ограничение оставляет возможность специальной обработки имен хорошо известных пакетов или путей в будущем.
> В Node.js можно использовать относительный URL-адрес, начинающийся с `./`, `../`, или URL со схемой `file://`. Можно также задать имя пакета.

10.4. Именованный импорт

Модуль может экспортировать именованные средства дополнительно или вместо умалчиваемого. Автор модуля дает имена всем не подразумеваемым по умолчанию средствам. Количество экспортируемых именованных средств не ограничено.

В примере ниже мы импортируем две функции, которые в модуле названы `encrypt` и `decrypt`:

```
import { encrypt, decrypt } from './modules/caesar.mjs'
```

Конечно, тут есть потенциальная опасность. Что, если мы хотим импортировать функции шифрования из двух модулей и в обоих они называются `encrypt`? По счастью, импортированные средства можно переименовывать:

```
import { encrypt as caesarEncrypt, decrypt as caesarDecrypt }
from './modules/caesar.mjs'
```

Таким образом, мы всегда можем избежать конфликта имен.

Если вы хотите импортировать и средство по умолчанию, и одно или несколько именованных средств, то обе синтаксические конструкции можно объединить:

```
import CaesarCipher, { encrypt, decrypt } from './modules/caesar.mjs'
```

или

```
import CaesarCipher, { encrypt as caesarEncrypt, decrypt as caesarDecrypt} ...
```

> **Примечание.** Не забывайте фигурные скобки при импорте одного именованного средства:
>
> ```
> import { encrypt } from './modules/caesar.mjs'
> ```
>
> Без скобок вы просто присвоите имя средству по умолчанию.

Если модуль экспортирует много имен, то было бы утомительно перечислять их все в предложении `import`. Вместо этого можно поместить все экспортированные средства в объект:

```
import * as CaesarCipherTools from './modules/caesar.mjs'
```

А затем обращаться к импортированным функциям по именам `CaesarCipherTools.encrypt` и `CaesarCipherTools.decrypt`. Если имеется средство по умолчанию, то оно будет доступно по имени `CaesarCipherTools.default`. Можно также поименовать его явно:

```
import CaesarCipher, * as CaesarCipherTools ...
```

Предложение `import` можно использовать, ничего не импортируя:

```
import './app/init.mjs'
```

Тогда предложения в файле выполняются, но ничего не импортируется. Эта возможность применяется редко.

10.5. Динамический импорт

На четвертой стадии рассмотрения сейчас находится предложение, которое позволит импортировать модуль, местоположение которого не фиксировано. Загрузка модуля по запросу может оказаться полезной для уменьшения времени запуска и потребления памяти приложением.

Для динамического импорта ключевое слово `import` используется так, будто это функция, которой в качестве аргумента передано местоположение модуля:

```
import(`./plugins/${action}.mjs`)
```

Предложение динамического импорта загружает модуль асинхронно. При этом возвращается обещание получить объект, содержащий все экспортированные средства. Обещание считается выполненным, когда модуль загружен. После этого его средства можно использовать:

```
import(`./plugins/${action}.mjs`)
  .then(module => {
```

```
    module.default()
    module.именованноеСредство(аргументы)
    ...
})
```

Разумеется, можно пользоваться нотацией `async/await`:

```
async load(action) {
  const module = await import(`./plugins/${action}.mjs`)
  module.default()
  module.именованноеСредство(аргументы)
  ...
}
```

При применении динамического импорта средства не импортируются по имени, и синтаксис для переименования средств отсутствует.

> **Примечание.** Ключевое слово `import` не является функцией, пусть даже выглядит похоже. Ему просто придана форма, подобная функции. Тут можно провести аналогию с ключевым словом `super` и синтаксической конструкцией `super(...)`.

10.6. Экспорт

Зная, как импортировать средства из модулей, рассмотрим модуль с точки зрения автора.

10.6.1. Именованный экспорт

Внутри модуля сколько угодно объявлений функций, классов и переменных можно пометить ключевым словом `export`:

```
export function encrypt(str, key) { ... }
export class Cipher { ... }
export const DEFAULT_KEY = 3
```

Можно вместо этого написать предложение `export`, содержащее имена экспортируемых средств:

```
function encrypt(str, key) { ... }
class Cipher { ... }
const DEFAULT_KEY = 3
...
export { encrypt, Cipher, DEFAULT_KEY }
```

Такая форма позволяет назначить экспортируемым средствам другие имена:

```
export { encrypt as caesarEncrypt, Cipher, DEFAULT_KEY }
```

Помните, что предложение `export` определяет имя, под которым средство экспортируется. Как мы видели ранее, импортирующий модуль может воспользоваться предложенным именем или выбрать свое.

> **Примечание.** Экспортируемые средства должны быть определены в области видимости верхнего уровня. Нельзя экспортировать локальные функции, классы или переменные.

10.6.2. Экспорт по умолчанию

Меткой `export default` может быть помечено не более одного класса или функции:

```
export default class Cipher { . . . }
```

Здесь средством модуля по умолчанию становится класс `Cipher`.

Метка `export default` неприменима к объявлениям переменных. Если вы хотите экспортировать по умолчанию значение, то не объявляйте его как переменную. Просто напишите `export default` и далее значение:

```
export default 3 // OK
export default const DEFAULT_KEY = 3
  // ошибка - export default неприменимо к объявлениям const/let/var
```

Маловероятно, что кому-то захочется делать простую константу экспортируемым по умолчанию значением. Гораздо более реалистичный сценарий – экспорт объекта, содержащего несколько средств:

```
export default { encrypt, Cipher, DEFAULT_KEY }
```

Этот синтаксис можно использовать вместе с анонимной функцией или классом:

```
export default (s, key) => { ... } // именовать функцию необязательно
```

или

```
export default class { // именовать класс необязательно
  encrypt(key) { ... }
  decrypt(key) { ... }
}
```

Наконец, для объявления средства по умолчанию можно использовать синтаксис переименования:

```
export { Cipher as default }
```

> **Примечание.** Средство по умолчанию – это просто средство с именем `default`. Но поскольку `default` – ключевое слово, оно не может быть идентификатором, так что необходимо использовать одну из синтаксических форм, описанных в этом разделе.

10.6.3. Экспортируемые средства – это переменные

Всякое экспортируемое средство является переменной, имеющей имя и значение. Значением может быть функция, класс и вообще произвольное значение JavaScript.

Значение экспортируемого средства может со временем изменяться. Эти изменения видны импортирующим модулям. Иными словами, экспортируемое средство захватывает именно переменную, а не просто ее значение.

Например, модуль протоколирования мог бы экспортировать переменную, содержащую текущий уровень протоколирования, и функцию для его изменения:

```
export const Level = { FINE: 1, INFO: 2, WARN: 3, ERROR: 4 }
export let currentLevel = Level.INFO
export const setLevel = level => { currentLevel = level }
```

Теперь рассмотрим модуль, который импортирует модуль протоколирования с помощью предложения:

```
import * as logging from './modules/logging.mjs'
```

Первоначально значение `logging.currentLevel` в этом модуле равно `Level.INFO`, или `2`. Если модуль вызывает метод

```
logging.setLevel(logging.Level.WARN)
```

то переменная `logging.currentLevel` изменяется и становится равна `3`.

Однако в импортирующем модуле эту переменную можно только читать. Нельзя положить

```
logging.currentLevel = logging.Level.WARN
  // Ошибка - присваивание импортированной переменной запрещено
```

Переменные, в которых хранятся экспортированные средства, создаются на этапе синтаксического анализа модуля, но заполняются только при выполнении тела модуля. Благодаря этому возможны циклические зависимости между модулями (см. упражнение 6).

> **Предостережение.** Если имеется кольцо модулей, зависящих друг от друга, то может случиться, что экспортируемое средство еще равно `undefined` к моменту, когда оно понадобилось другому модулю (см. упражнение 11).

10.6.4. Реэкспорт

Модуль с развитым API и сложной реализацией, скорее всего, будет зависеть от других модулей. Конечно, система модулей берет на себя управление зависимостями, так что пользователю модуля об этом думать не надо. Но может случиться так, что какой-то модуль содержит полезные средства, которые вы

хотите предоставить в распоряжение своих пользователей. Тогда можно не требовать от пользователей, чтобы они импортировали эти средства самостоятельно, а *реэкспортировать* их.

Вот как реэкспортируются средства из другого модуля:

```
export { randInt, randDouble } from './modules/random.mjs'
```

Всякий, кто импортирует этот модуль, получит также средства `randInt` и `randDouble` из модуля `'./modules/random.mjs'`, как будто они определены в этом модуле.

При желании можно переименовать реэкспортируемые средства:

```
export { randInt as randomInteger } from './modules/random.mjs'
```

Чтобы реэкспортировать из модуля средство по умолчанию, обратитесь к нему по имени `default`:

```
export { default } from './modules/stringutil.mjs'
export { default as StringUtil } from './modules/stringutil.mjs'
```

Можно, наоборот, реэкспортировать какое-то средство, сделав его средством по умолчанию в своем модуле:

```
export { Random as default } from './modules/random.mjs'
```

Наконец, можно реэкспортировать все средства из другого модуля, кроме средства по умолчанию:

```
export * from './modules/random.mjs'
```

Это имеет смысл делать, если вы разбиваете свой проект на много мелких модулей, а затем предоставляете единый модуль в качестве фасада, реэкспортируя все составляющие модули.

Предложение `export *` пропускает средство по умолчанию, потому что попытка реэкспортировать средства по умолчанию из нескольких модулей привела бы к конфликту.

10.7. Упаковка модулей

Модули отличаются от простых «скриптов»:

- код внутри модуля всегда выполняется в строгом режиме;
- у каждого модуля имеется собственная область видимости верхнего уровня, отличная от глобальной области видимости исполняющей среды JavaScript;
- модуль обрабатывается только один раз, даже если загружается несколько раз;
- модуль обрабатывается асинхронно;
- модуль может содержать предложения `import` и `export`.

Читая содержимое модуля, исполняющая среда JavaScript должна знать, что обрабатывает модуль, а не простой скрипт.

В браузере модуль загружается с помощью тега `script` с атрибутом `type`, равным `module`.

```
<script type="module" src="./modules/caesar.mjs"></script>
```

В Node.js можно использовать расширение имени файла `.mjs`, чтобы показать, что файл является модулем. Если вы хотите оставить обычное расширение `.js`, то должны помечать модули в конфигурационном файле `package.json`. При запуске исполняемого файла `node` в интерактивном режиме задавайте параметр командной строки `--input-type=module`.

По-видимому, самое простое – всегда использовать для модулей расширение `.mjs`. Его распознают все исполняющие среды и инструменты сборки.

> **Примечание.** Чтобы веб-сервер правильно отдавал файлы с расширением `.mjs`, его нужно настроить так, чтобы в ответе присутствовал заголовок `Content-Type: text/javascript`.

> **Предостережение.** В отличие от обычных скриптов, при загрузке модулей браузеры применяют ограничение CORS. Если вы загружаете модули из другого домена, то сервер должен возвращать заголовок `Access-Control-Allow-Origin`.

> **Примечание.** На третьей стадии рассмотрения находится предложение ввести объект `import.meta`, который будет предоставлять информацию о текущем модуле. Некоторые исполняющие среды JavaScript раскрывают URL-адрес, с которого был загружен модуль в виде свойства `import.meta.url`.

Упражнения

1. Найдите JavaScript-библиотеку для статистических расчетов (например, https://github.com/simple-statistics/simple-statistics). Напишите программу, которая импортирует эту библиотеку в виде модуля ECMAScript и вычисляет среднее и стандартное отклонение некоторого набора данных.
2. Найдите JavaScript-библиотеку для шифрования (например, https://github.com/brix/crypto-js). Напишите программу, которая импортирует эту библиотеку в виде модуля ECMAScript и шифрует, а затем дешифрует сообщение.
3. Напишите простой модуль протоколирования, который поддерживает протоколирование сообщений, для которых уровень протоколирования превышает заданный порог. Экспортируйте функцию `log`, константы уровней протоколирования и функцию для задания порога.
4. Повторите предыдущее упражнение, но экспортируйте весь класс в качестве средства по умолчанию.
5. Напишите простой модуль шифрования, в котором используется шифр Цезаря (прибавление константы к каждой кодовой точке). Воспользуйтесь модулем протоколирования из предыдущих упражнений, чтобы протоколировать все обращения к `decrypt`.

6. В качестве примера циклической зависимости между модулями повторите предыдущее упражнение, но предоставьте возможность шифровать протоколируемые сообщения.
7. Напишите простой модуль, который предоставляет случайные целые числа, массивы случайных целых чисел и случайные строки. Используйте как можно больше синтаксических форм `export`.
8. В чем различие между

```
import Cipher from './modules/caesar.mjs'
```

и

```
import { Cipher } from './modules/caesar.mjs'
```

9. В чем различие между

```
export { encrypt, Cipher, DEFAULT_KEY }
```

и

```
export default { encrypt, Cipher, DEFAULT_KEY }
```

10. Какие из следующих предложений допустимы в JavaScript?

```
export function default(s, key) { ... }
export default function (s, key) { ... }
export const default = (s, key) => { ... }
export default (s, key) => { ... }
```

11. В деревьях встречаются узлы двух видов: имеющие потомков (родители) и не имеющие потомков (листья). Смоделируем эту ситуацию с помощью наследования:

```
class Node {
 static from(value, ...children) {
   return children.length === 0 ? new Leaf(value)
     : new Parent(value, children)
 }
}

class Parent extends Node {
 constructor(value, children) {
   super()
   this.value = value
   this.children = children
 }
 depth() {
   return 1 + Math.max(...this.children.map(c => c.depth()))
 }
}

class Leaf extends Node {
 constructor(value) {
```

```
    super()
    this.value = value
  }
  depth() {
    return 1
  }
}
```

Теперь разработчик, обожающий модули, хочет поместить каждый класс в отдельный модуль. Сделайте это и протестируйте с помощью демонстрационной программы:

```
import { Node } from './node.mjs'
const myTree = Node.from('Adam',
 Node.from('Cain', Node.from('Enoch')),
 Node.from('Abel'),
 Node.from('Seth', Node.from('Enos')))
console.log(myTree.depth())
```

Что происходит? Почему?

12. Конечно, проблемы, имеющей место в предыдущем упражнении, можно было бы легко избежать, если не пользоваться наследованием и поместить все классы в один модуль. Но в более крупной системе такой возможности может и не представиться. Оставьте каждый класс в своем модуле и добавьте модуль-фасад `tree.mjs`, который реэкспортирует все три модуля. Во всех модулях импортируйте `'./tree.mjs'`, а не отдельные модули. Почему такой подход решает проблему?

Глава 11

Метапрограммирование

В этой главе мы рассмотрим более сложные API, которые позволяют создавать объекты с нестандартным поведением и писать код, работающий с обобщенными объектами.

Мы начнем с символов[1] – единственного типа, кроме строк, который можно использовать для именования свойств объектов. Определяя свойства с помощью «хорошо известных» символов, мы можем настраивать поведение некоторых методов API.

Затем мы внимательно рассмотрим свойства объектов. У свойств могут быть атрибуты, и мы узнаем, как анализировать, создавать и изменять свойства с помощью подходящих атрибутов. В качестве примера разберем функцию клонирования, которая умеет создавать глубокие копии.

После этого мы обратимся к объектам-функциям и методам для привязки параметров и вызова функций с заданными параметрами. Наконец, увидим, как с помощью прокси можно перехватывать все аспекты работы с объектами. Мы подробно рассмотрим два приложения: наблюдение за доступом к объектам и динамическое создание свойств.

11.1. Символы

В этой книге мы не раз видели, что объект в JavaScript имеет ключи типа `String`. Однако у использования строк в качестве ключей есть ограничения. В современном JavaScript предлагается еще один тип, пригодный для определения ключей объектов, – `Symbol`.

У символов имеются строковые метки, но сами они строками не являются. Символ создается следующим образом:

```
const sym = Symbol('label')
```

---

[1] Исторически сложилось, что словом «символ» переводятся слова «character» (которое в этой книге обычно переводится как «литера» или «знак», за исключением употребления в устойчивых словосочетаниях) и «symbol». В этой главе под «символом» понимается именно «symbol». – *Прим. перев.*

Символы уникальны. Если создать второй символ

```
const sym2 = Symbol('label')
```

то `sym !== sym2`.

В этом заключается принципиальное преимущество символов. Если вам требуется гарантированно уникальный строковый ключ, то можно добавить счетчик, или временную метку, или случайное число, но все равно остается тревога – достаточно ли этого?

> **Примечание.** Использовать ключевое слово `new` для создания символа нельзя: `new Symbol('label')` возбуждает исключение.

Поскольку символы не являются строками, для символьных ключей нельзя использовать точку. Вместо этого пользуйтесь квадратными скобками:

```
let obj = { [sym]: initialValue }
obj[sym] = newValue
```

Если требуется присоединить свойство к существующему объекту, например узлу DOM, то лучше не пользоваться строковым ключом:

```
node.outcome = 'success'
```

Даже если у узлов пока нет ключа с именем `outcome`, он может появиться в будущей версии.

Но использовать для этой цели символ абсолютно безопасно:

```
let outcomeSymbol = Symbol('outcome')
node[outcomeSymbol] = 'success'
```

Заметим, что символ необходимо сохранить в переменной или объекте, чтобы он был доступен, когда понадобится.

Например, в классе `Symbol` встречаются «хорошо известные» символы в полях `Symbol.iterator` и `Symbol.species`, которые мы изучим в следующем разделе.

Если необходимо использовать одни и те же символы в нескольких «областях» (например, в разных IFRAME или рабочих веб-процессах), то можно прибегнуть к глобальному реестру символов. Чтобы создать или получить ранее созданный глобальный символ, вызовите метод `Symbol.for` и укажите ключ, который должен быть глобально уникальным:

```
let sym3 = Symbol.for('com.horstmann.outcome')
```

> **Примечание.** Оператор `typeof`, примененный к символу, возвращает строку `'symbol'`.

11.2. Настройка с помощью символьных свойств

Символьные свойства используются в JavaScript API для настройки поведения классов. В классе `Symbol` определен ряд «хорошо известных» символьных

констант для этой цели. Все они перечислены в табл. 11.1, и три из них мы рассмотрим ниже.

Таблица 11.1. Хорошо известные символы

Символ	Описание
`toStringTag`	Настраивает метод `toString` класс `Object`, см. раздел 11.2.1
`toPrimitive`	Настраивает преобразование в примитивный тип, см. раздел 11.2.2
`species`	Функция-конструктор для создания результирующей коллекции. Используется в таких методах, как `map` и `filter`, см. раздел 11.2.2
`iterator`, `asyncIterator`	Определяет итераторы (глава 9) и асинхронные итераторы (глава 10)

`hasInstance`	Настраивает поведение `instanceof`:
	`class Iterable {`
	`  static [Symbol.hasInstance](obj) {`
	`  return Symbol.iterator in obj`
	`  }`
	`}`
	`[1, 2, 3] instanceof Iterable`

`match`, `matchAll`, `replace`, `search`, `split`	Вызываются из одноименных методов класса `String`. Переопределены для объектов, отличных от `RegExp`, см. упражнение 2

`isConcatSpreadable`	Используется в методе `concat` класса `Array`:
	`const a = [1, 2]`
	`const b = [3, 4]`
	`a[Symbol.isConcatSpreadable] = false`
	`[].concat(a, b) ⇒ [[1, 2], 3, 4]`

11.2.1. Настройка метода toString

Можно изменить поведение метода `toString` в классе `Object`. По умолчанию он возвращает `'[object Object]'`. Но если у объекта имеется свойство с ключом `Symbol.toStringTag`, то вместо `Object` используется значение этого свойства. Например:

```
const harry = { name: 'Harry Smith', salary: 100000 }
harry[Symbol.toStringTag] = 'Employee'
console.log(harry.toString())
  // Теперь toString возвращает '[object Employee]
```

При определении класса можно установить это свойство в конструкторе:

```
class Employee {
  constructor(name, salary) {
    this[Symbol.toStringTag] = 'Employee'
    ...
  }
  ...
}
```

Или завести метод `get` со специальным синтаксисом:

```
class Employee {
  ...
  get [Symbol.toStringTag]() { return JSON.stringify(this) }
}
```

Идея в том, что хорошо известный символ определяет точку для настройки поведения метода API.

11.2.2. Управление преобразованием типов

Символ `Symbol.toPrimitive` дает дополнительный контроль над преобразованием в примитивные типы в случае, когда переопределения метода `valueOf` недостаточно. Рассмотрим следующий класс, представляющий проценты:

```
class Percent {
  constructor(rate) { this.rate = rate }
  toString() { return `${this.rate}%` }
  valueOf() { return this.rate * 0.01 }
}
```

Теперь рассмотрим такой код:

```
const result = new Percent(99.44)
console.log('Result: ' + result) // печатается 0.9944
```

Почему не `'99.44%'`? Потому что оператор `+` пользуется методом `valueOf`, если тот имеется. Для исправления ситуации нужно добавить метод с ключом `Symbol.toPrimitive`:

```
[Symbol.toPrimitive](hint) {
  if (hint === 'number') return this.rate * 0.01
  else return `${this.rate}%`
}
```

Параметр `hint` может принимать следующие значения:

- `'number'` для арифметических операций, кроме сложения и сравнений;
- `'string'` для `` `${...}` `` или `String(...)`;
- `'default'` для `+` или `==`.

На практике ценность этого механизма ограничена, потому что `hint` не дает достаточной информации. В действительности нам нужен тип другого операнда (см. упражнение 1).

11.2.3. Символ Species

По умолчанию метод `map` класса `Array` порождает коллекцию того же типа, которую получил:

```
class MyArray extends Array {}
let myValues = new MyArray(1, 2, 7, 9)
myValues.map(x => x * x) // возвращает MyArray
```

Это не всегда то, что нужно. Пусть имеется класс Range, расширяющий Array и предназначенный для описания диапазона целых чисел:

```
class Range extends Array {
  constructor(start, end) {
    super()
    for (let i = 0; i < end - start; i++)
      this[i] = start + i
  }
}
```

Результат преобразования диапазона обычно не является диапазоном:

```
const myRange = new Range(10, 99)
myRange.map(x => x * x) // это не Range
```

В таком классе коллекции можно задать другой конструктор, являющийся значением свойства Symbol.species:

```
class Range extends Array {
  ...
  static get [Symbol.species]() { return Array }
}
```

Эта функция-конструктор используется в методах класса Array, создающих новые массивы: map, filter, flat, flatMap, subarray, slice, splice, concat.

11.3. Атрибуты свойств

В этом и следующих разделах мы рассмотрим все функции и методы класса Object, перечисленные в табл. 11.2.

Таблица 11.2. Функции и методы класса Object

Имя	Описание
Функции	
defineProperty(obj, name, descriptor) defineProperties(obj, { name1: descriptor1, ... })	Определить один или несколько дескрипторов свойств
getOwnPropertyDescriptor(obj, name) getOwnPropertyDescriptors(obj) getOwnPropertyNames(obj) getOwnPropertySymbols(obj)	Получить один или все неунаследованные дескрипторы объекта либо только их строковые имена или символы
keys(obj) values(obj) entries(obj)	Имена, значения и пары [имя, значение] собственных перечисляемых свойств
preventExtensions(obj) seal(obj) freeze(obj)	Запретить изменение прототипа и добавление свойств; также удаление и конфигурирование свойств; также изменение свойств

Таблица 11.2 *(окончание)*

Имя	Описание

`isExtensible(obj)`
`isSealed(obj)`
`isFrozen(obj)`
	Проверить, защищен ли объект одной из функций в предыдущей строке

`create(prototype, { name1: descriptor1, ... })` `fromEntries([[name1, value1], ...])`	Создать новый объект с заданными свойствами
`assign(target, source1, source2, ...)`	Скопировать все перечисляемые собственные свойства из исходных объектов в целевой
`getPrototypeOf(obj)` `setPrototypeOf(obj, proto)`	Получить или установить прототип
Методы	
`hasOwnProperty(stringOrSymbol)` `propertyIsEnumerable(stringOrSymbol)`	`true`, если объект обладает указанным свойством или является перечисляемым
`isPrototypeOf(other)`	Проверить, является ли этот объект прототипом объекта `other`

Начнем с того, что изучим работу со свойствами объекта. У каждого свойства JavaScript-объекта есть три *атрибута*:

1) `enumerable`: если `true`, то это свойство посещается в цикле `for in`;
2) `writable`: если `true`, то это свойство можно обновлять;
3) `configurable`: если `true`, то это свойство можно удалять, а его атрибуты модифицировать.

Если свойство задано в объектном литерале или путем присваивания, то все три атрибута равны `true`, с одним исключением: свойства с символьными ключами не являются перечисляемыми.

```
let james = { name: 'James Bond' }
  // свойство james.name допускает запись, перечисление и конфигурирование
```

С другой стороны, свойство `length` массива допускает запись, но не является ни перечисляемым, ни конфигурируемым.

> **Примечание.** В строгом режиме действия, нарушающие предписания атрибутов `writable` и `configurable`, приводят к исключению. В нестрогом режиме они молчаливо игнорируются.

Мы можем динамически определять свойства с произвольными именами и значениями атрибутов, вызвав функцию `Object.defineProperty`:

```
Object.defineProperty(james, 'id', {
  value: '007',
  enumerable: true,
  writable: false,
  configurable: true
})
```

Последний аргумент называется *дескриптором свойств*.

Если при определении нового свойства какой-то атрибут не задан, то ему присваивается значение `false`.

Ту же самую функцию можно использовать для изменения атрибутов существующего свойства, если только оно конфигурируемое:

```
Object.defineProperty(james, 'id', {
  configurable: false
}) // Теперь james.id нельзя удалить, а его атрибуты изменить
```

Мы можем определить акцессоры чтения и записи, предоставив функции с ключами `get` и `set`:

```
Object.defineProperty(james, 'lastName', {
  get: function() { return this.name.split(' ')[1] },
  set: function(last) { this.name = this.name.split(' ')[0] + ' ' + last }
})
```

Отметим, что использовать здесь стрелочные функции нельзя, потому что нам нужен параметр `this`.

Функция `get` вызывается, когда свойство используется как значение:

```
console.log(james.lastName) // печатается Bond
```

Функция `set` вызывается, когда свойству присваивается новое значение:

```
james.lastName = 'Smith' // теперь james.name равно 'James Smith'
```

> **Примечание.** В главе 4 мы видели, как определяются акцессоры чтения и записи в классе: путем добавления к имени метода префикса `get` или `set`. А теперь мы знаем, что для их определения и класс-то не нужен.

Наконец, функция `Object.defineProperties` позволяет определить или обновить несколько свойств. Ей нужно передать объект, ключами которого являются имена свойств, а значениями – дескрипторы свойств.

```
Object.defineProperties(james, {
  id: { value: '007', writable: false, enumerable: true, configurable: false },
  age: { value: 42, writable: true, enumerable: true, configurable: true }
})
```

11.4. Перечисление свойств

В предыдущем разделе мы видели, как определить одно или несколько свойств. Функции `getOwnPropertyDescriptor` и `getOwnPropertyDescriptors` возвращают дескрипторы свойств в том же формате, в каком представлены аргументы функций `defineProperty` и `defineProperties`. Например,

```
Object.getOwnPropertyDescriptor(james, 'name')
```

возвращает дескриптор

```
{ value: 'James Bond',
  writable: true,
```

```
  enumerable: true,
  configurable: true }
```

Для получения всех дескрипторов нужно вызвать

```
Object.getOwnPropertyDescriptors(james)
```

В ответ возвращается объект, ключами которого являются имена свойств, а значениями – соответствующие дескрипторы:

```
{ name:
    { value: 'James Bond',
      writable: true,
      enumerable: true,
      configurable: true },
  lastName:
    { get: [Function: get],
      set: [Function: set],
      enumerable: false,
      configurable: false }
  ...
}
```

Функция называется `getOwnPropertyDescriptors`, потому что возвращает только свойства, определенные в самом объекте, а не унаследованные по цепочке прототипов.

> **Совет.** Функция `Object.getOwnPropertyDescriptors` очень полезна для инспекции объекта, поскольку возвращает все свойства, в т. ч. неперечисляемые (см. упражнение 9).

Если вам не нужна обширная информация, которую возвращает `Object.getOwnPropertyDescriptors`, то можете вызвать `Object.getOwnPropertyNames(obj)` или `Object.getOwnPropertySymbols(obj)` для получения ключей свойств, представленных строками или символами (в т. ч. неперечисляемых), а затем найти те дескрипторы, которые вас интересуют.

Наконец, функции `Object.keys`, `Object.values` и `Object.entries` возвращают соответственно имена, значения и пары [*имя*, *значение*] для собственных перечисляемых свойств. Они аналогичны методам `keys`, `values` и `entries` класса `Map`, с которыми мы встречались в главе 7. Однако это не методы, и возвращают они массивы, а не итераторы.

```
const obj = { name: 'Fred', age: 42 }
Object.entries(obj) // [['name', 'Fred'], ['age', 42]]
```

Перебрать все свойства можно в таком цикле:

```
for (let [key, value] of Object.entries(obj))
  console.log(key, value)
```

11.5. Проверка наличия свойства

Условие

```
stringOrSymbol in obj
```

проверяет, существует ли свойство в самом объекте или в одном из его прототипов.

Почему бы просто не проверить, что `obj[stringOrSymbol] !== undefined`? Оператор `in` возвращает `true` для свойств со значением `undefined`. Для объекта

```
const harry = { name: 'Harry', partner: undefined }
```

условие `'partner' in harry` равно `true`.

Иногда мы не хотим просматривать цепочку прототипов. Чтобы узнать, есть ли в самом объекте свойство с заданным именем, вызовите метод

```
obj.hasOwnProperty(stringOrSymbol)
```

Чтобы проверить наличие перечисляемого свойства, вызовите

```
obj.propertyIsEnumerable(stringOrSymbol)
```

Заметим, что у этих методов есть потенциальный недостаток. Объект может переопределять методы и лгать о своих свойствах. В этом смысле безопаснее использовать оператор `in` функции типа `Object.getOwnPropertyDescriptior`.

11.6. Защита объектов

В классе `Object` есть три функции для защиты объектов. Перечислим их в порядке усиления защиты.

1. `Object.preventExtensions(obj)`: нельзя добавлять собственные свойства и изменять прототип.
2. `Object.seal(obj)`: дополнительно нельзя удалять и конфигурировать свойства.
3. `Object.freeze(obj)`: дополнительно нельзя присваивать значения свойствам.

Все три функции возвращают защищаемый объект. Например, можно сконструировать и заморозить (freeze) объект следующим образом:

```
const frozen = Object.freeze({ ... })
```

Заметим, что эти средства защиты можно применять только в строгом режиме.

Даже заморозка не делает объект полностью защищенным от изменений, потому что значения свойств могут быть изменяемыми:

```
const fred = Object.freeze({ name: 'Fred', luckyNumbers: [17, 29] })
fred.luckyNumbers[0] = 13 // OK - массив luckyNumbers не заморожен
```

Если вам нужна полная неизменяемость, то придется рекурсивно заморозить все вложенные объекты (см. упражнение 8).

Чтобы узнать, был ли объект защищен с помощью одной из этих функций, можно воспользоваться функциями `Object.isExtensible(obj)`, `Object.isSealed(obj)` или `Object.isFrozen(obj)`.

11.7. Создание и обновление объектов

Функция `Object.create` дает полный контроль над созданием нового объекта. Задайте прототип, а также имена и дескрипторы всех свойств:

```
const obj = Object.create(proto, propertiesWithDescriptors)
```

Здесь `propertiesWithDescriptors` – объект, ключами которого являются имена свойств, а значениями – дескрипторы, как в разделе 11.4.

Если имена и значения свойств помещены в итерируемый объект, состоящий из массивов [*ключ*, *значение*], то можно вызвать функцию `Object.fromEntries`, которая создаст объект с такими свойствами:

```
let james = Object.fromEntries([['name', 'James Bond'], ['id', '007']])
```

Вызов `Object.assign(target, source1, source2, ...)` копирует все собственные перечисляемые свойства объектов-источников в целевой объект и возвращает обновленный целевой объект:

```
james = Object.assign(james, { salary: 300000 }, genericSpy)
```

В настоящее время нет никаких причин использовать функцию `Object.assign`. Расширение удобнее:

```
{ ...james, salary: 300000, ...genericSpy }.
```

11.8. Доступ к прототипу и его обновление

Мы уже знаем, что цепочка прототипов – ключевое понятие при программировании на JavaScript. Если пользоваться ключевыми словами `class` и `extends`, то цепочка прототипов строится автоматически. В этом разделе мы расскажем, как управлять ей вручную.

Чтобы получить прототип объекта (т. е. значение внутреннего слота [[Prototype]]), нужно вызвать

```
const proto = Object.getPrototypeOf(obj)
```

Например:

```
Object.getPrototypeOf('Fred') === String.prototype
```

Если имеется экземпляр класса, созданного оператором `new`, например

```
const obj = new ClassName(args)
```

то результат вызова `Object.getPrototypeOf(obj)` совпадает с `ClassName.prototype`. Но прототип любого объекта можно установить, вызвав

```
Object.setPrototypeOf(obj, proto)
```

Мы уже кратко описывали этот способ в главе 4, перед тем как познакомились с оператором `new`.

Однако изменение прототипа существующего объекта – медленная операция в виртуальных машинах JavaScript, потому что они рассчитаны на то, что прототипы объектов не изменяются. Если необходимо создать объект со специальным прототипом, то лучше использовать метод `Object.create`, описанный в разделе 11.7.

Вызов `proto.isPrototypeOf(obj)` возвращает `true`, если `proto` встречается в цепочке прототипов объекта `obj`. Если не был задан специальный прототип, то можно воспользоваться просто оператором `instanceof`: `obj instanceof ClassName` – то же самое, что `ClassName.prototype.isPrototypeOf(obj)`.

> **Примечание.** В отличие от всех остальных объектов-прототипов, `Array.prototype` действительно является массивом!

11.9. Клонирование объектов

В качестве примера применения материала, изложенного в предыдущих разделах, разработаем функцию, которая умеет делать глубокую копию объекта (клонировать его).

Наивный подход – воспользоваться оператором расширения:

```
const cloned = { ...original } // в общем случае это не полный клон
```

Однако при этом копируются только перечисляемые свойства. И совсем не учитываются прототипы.

Мы можем скопировать прототип и все его свойства:

```
const cloned = Object.create(Object.getPrototypeOf(original),
Object.getOwnPropertyDescriptors(original)) // лучше, но все равно поверхностно
```

Теперь клон имеет тот же прототип и те же свойства, что оригинал, и все атрибуты свойств скопированы в точности.

Но копия все равно поверхностная. Значения изменяемых свойств не клонируются. Чтобы понять, в чем проблема, рассмотрим следующий объект:

```
const original = { radius: 10, center: { x: 20, y: 30 } }
```

Тогда `original.center` и `clone.center` – это один и тот же объект, как видно по рис. 11.1. При изменении `original` меняется также и `clone`:

```
original.center.x = 40 // clone.center.x также изменился
```

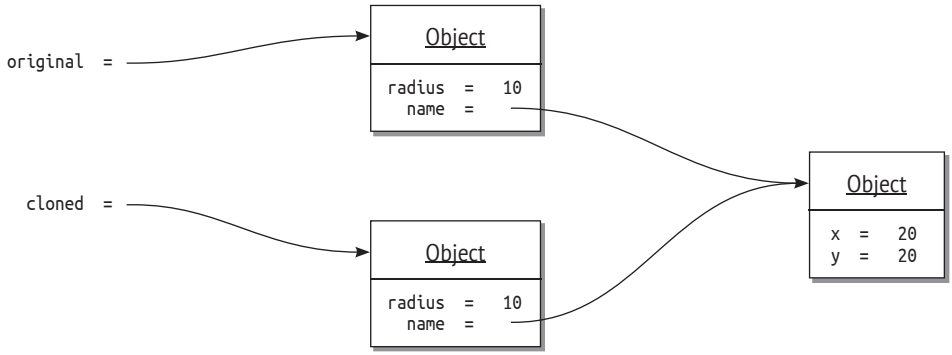

Рис. 11.1 ❖ Поверхностная копия

Чтобы исправить ошибку, нужно рекурсивно клонировать все значения:

```
const clone = obj => {
  if (typeof obj !== 'object' || Object.isFrozen(obj)) return obj
  const props = Object.getOwnPropertyDescriptors(obj)
  let result = Object.create(Object.getPrototypeOf(obj), props)
  for (const prop in props)
    result[prop] = clone(obj[prop])
  return result
}
```

Однако эта функция не справляется с циклическими ссылками.

Рассмотрим двух людей, каждый из которых является лучшим другом другого (см. рис. 11.2):

```
const fred = { name: 'Fred' }
const barney = { name: 'Barney' }
fred.bestFriend = barney
barney.bestFriend = fred
```

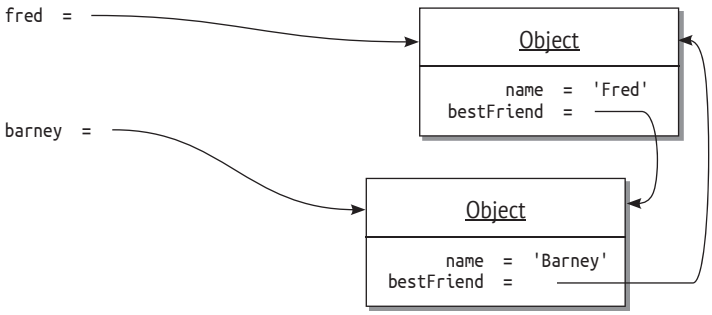

Рис. 11.2 ❖ Циклические ссылки

Теперь предположим, что мы рекурсивно клонируем объект `fred`. Результатом будет новый объект

```
cloned = { name: 'Fred', bestFriend: clone(barney) }
```

А что делает вызов `clone(barney)`? Создает объект `{ name: 'Barney', bestFriend: clone(fred) }`. Однако это никуда не годится – налицо бесконечная рекурсия. Но даже если бы это было не так, мы получили бы объект с неправильной структурой. Мы ожидаем, что

```
cloned.bestFriend.bestFriend === cloned
```

Необходимо улучшить процесс рекурсивного клонирования. Если объект уже был клонирован, надо не клонировать его снова, а использовать ссылку на существующий клон. Это можно реализовать, построив отображение оригиналов на клоны. Если встречается еще не клонированный объект, то надо добавить ссылки на оригинал и клон в отображение. А если объект уже был клонирован, то достаточно найти клон.

```
const clone = (obj, cloneRegistry = new Map()) => {
  if (typeof obj !== 'object' || Object.isFrozen(obj)) return obj
  if (cloneRegistry.has(obj)) return cloneRegistry.get(obj)
  const props = Object.getOwnPropertyDescriptors(obj)
  let result = Object.create(Object.getPrototypeOf(obj), props)
  cloneRegistry.set(obj, result)
  for (const prop in props)
    result[prop] = clone(obj[prop], cloneRegistry)
  return result
}
```

Это уже очень близко к правильной функции клонирования. Однако она не работает для массивов. Вызвав `clone([1, 2, 3])`, мы получим массивоподобный объект с прототипом `Array.prototype`. Однако это не массив – `Array.isArray` возвращает `false`.

Чтобы исправить ошибку, нужно копировать массивы с помощью функции `Arrays.from`, а не `Object.create`. Вот как выглядит окончательная версия:

```
const clone = (obj, cloneRegistry = new Map()) => {
  if (typeof obj !== 'object' || Object.isFrozen(obj)) return obj
  if (cloneRegistry.has(obj)) return cloneRegistry.get(obj)
  const props = Object.getOwnPropertyDescriptors(obj)
  let result = Array.isArray(obj) ? Array.from(obj)
    : Object.create(Object.getPrototypeOf(obj), props)
  cloneRegistry.set(obj, result)
  for (const prop in props)
    result[prop] = clone(obj[prop], cloneRegistry)
  return result
}
```

11.10. Свойства-функции

Обсудив методы класса `Object`, обратимся к объектам-функциям. У любой функции, являющейся экземпляром класса `Function`, имеется три неперечисляемых свойства:

1) `name`: имя, данное функции при определении, а для анонимных функций – имя переменной, которой была присвоена функция (см. упражнение 14);
2) `length`: количество аргументов, не считая аргумента «прочие»;
3) `prototype`: объект, который должен быть заполнен свойствами прототипа.

Вспомним, что в классическом `JavaScript` нет различий между функциями и конструкторами. Даже в строгом режиме любую функцию можно вызвать с помощью `new`. Поэтому у каждой функции имеется объект `prototype`.

Рассмотрим объект `prototype` функции более пристально. У него нет перечисляемых свойств, а единственное неперечисляемое свойство `constructor` указывает обратно на функцию-конструктор (см. рис. 11.3). Пусть, например, мы определяем класс `class Employee`. Функция-конструктор `Employee`, как и любая другая функция, имеет свойство `prototype` и

```
Employee.prototype.constructor === Employee
```

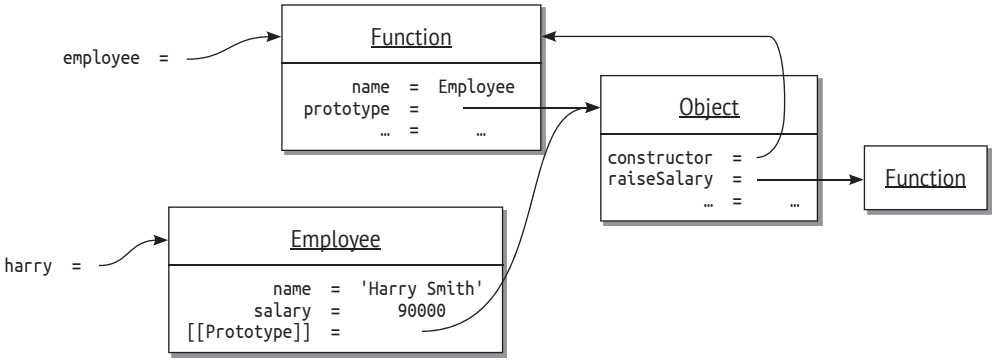

Рис. 11.3 ❖ Свойство `constructor`

Любой объект наследует свойство `constructor` от прототипа. Таким образом, мы можем получить имя класса объекта в виде

```
obj.constructor.name
```

> **Примечание.** Внутри конструктора результатом вычисления странного на вид выражения `new.target` является функция, с помощью которой сконструирован объект. Это выражение можно использовать, чтобы узнать, конструируется ли объект как экземпляр подкласса; иногда это бывает полезно (см. упражнение 11). Можно также сказать, была ли функция вызвана без `new`. В таком случае `new.target === undefined`.

11.11. Привязка аргументов и вызов методов

Если дана функция, то метод bind возвращает другую функцию, в которой начальные аргументы фиксированы:

```
const multiply = (x, y) => x * y
const triple = multiply.bind(null, 3)
triple(14) // возвращает 42, или multiply(3, 14)
```

Поскольку один аргумент multiply фиксирован методом bind, результатом является функция triple (утроить) с одним аргументом.

Первый аргумент метода bind – привязка параметра this, например:

```
const isPet = Array.prototype.includes.bind(['cat', 'dog', 'fish'])
```

Метод bind позволяет преобразовать метод в функцию:

```
button.onclick = this.handleClick.bind(this)
```

Никакой необходимости использовать bind в любом из рассмотренных случаев нет. Можно определить функции явно:

```
const triple = y => multiply(3, y)
const isPet = x => ['cat', 'dog', 'fish'].includes(x)
button.onclick = (...args) => this.handleClick(...args)
```

Метод call похож на bind. Однако передаются все аргументы, и функция или метод вызывается. Например:

```
let answer = multiply.call(null, 6, 7)
let uppercased = String.prototype.toUpperCase.call('Hello')
```

Конечно, гораздо проще вызвать multiply(6, 7) или 'Hello'.toUpperCase().

Однако есть одна ситуация, когда прямой вызов функции не работает. Рассмотрим пример:

```
const spacedOut = Array.prototype.join.call('Hello', ' ') // 'H e l l o'
```

Мы не можем вызвать

```
'Hello'.join(' ')
```

потому что join – не метод класса String. Это метод класса Array, который, по стечению обстоятельств, умеет работать со строками.

Наконец, метод apply похож на call, но все аргументы, кроме this, помещены в массив (или массивоподобный объект):

```
String.prototype.substring.apply('Hello', [1, 4]) // 'ell'
```

Если требуется применить произвольную функцию, хранящуюся в переменной f, к произвольным аргументам, то проще использовать выражение f(...args), а не f.apply(null, args). Но если в переменной f хранится *метод*,

то никакого выбора нет. Нельзя вызвать `obj.f(...args)`, приходится писать `f.apply(obj, args)`.

> **Примечание.** Раньше, когда в JavaScript не было ключевого слова `super`, мы вынуждены были использовать `bind`, `call` или `apply` для вызова конструктора суперкласса (см. упражнение 16).

11.12. Прокси

Прокси – это нечто такое, что пользователю представляется объектом, но на самом деле перехватывает доступ к свойствам, доступ к прототипу и обращения к методам. Перехваченные действия можно заменить чем угодно.

Например, ORM (объектно-реляционное отображение) может поддержать имена методов вида

```
const result = orm.findEmployeeById(42)
```

где `Employee` соответствует таблице базы данных. А если подходящей таблицы нет, то такой метод завершится ошибкой.

Здесь `orm` – прокси-объект, который перехватывает все обращения к методам. При вызове от его имени метода с именем вида `find...ById` перехватывающий код выделяет из имени метода имя таблицы и обращается к базе данных.

Это весьма мощная идея, позволяющая создавать динамичные и интересные эффекты. Приведем лишь несколько примеров:

- автоматическое протоколирование доступа к свойствам или изменения свойств;
- контроль доступа к свойствам, например проверка правильности или защита секретных данных;
- динамические свойства, например элементы DOM или столбцы таблицы базы данных;
- выполнение удаленных вызовов так, будто они локальны.

Для конструирования прокси нужны два объекта:

1) *целевой объект*, операции которого мы хотим контролировать;
2) *обработчик*, т. е. объект, содержащий *функции-перехватчики*, которые вызываются при операциях с прокси.

Тринадцать существующих функций-перехватчиков перечислены в табл. 11.3.

Начнем с простого примера – протоколирования операций чтения и записи свойств объекта `obj`. В обработчике установим два перехватчика:

```
const obj = { name: 'Harry Smith', salary: 100000 }
const logHandler = {
  get(target, key, receiver) {
    const result = target[key]
    console.log(`get ${key.toString()} as ${result}`)
    return result
  },
```

```
  set(target, key, value, receiver) {
    console.log(`set ${key.toString()} to ${value}`)
    target[key] = value
    return true
  }
}
const proxy = new Proxy(obj, logHandler)
```

Таблица 11.3. Функции-перехватчики

Имя	Описание
`get(target, key, receiver)`	`receiver[key]`, `receiver.key`
`set(target, key, value, receiver)`	`receiver[key] = value`, `receiver.key = value`
`deleteProperty(target, key)`	`delete proxy[key]`, `delete proxy.key`
`has(target, key)`	`key in target`
`getPrototypeOf(target)`	`Object.getPrototypeOf(proxy)`
`setPrototypeOf(target, proto)`	`Object.setPrototypeOf(proxy, proto)`
`isExtensible(target)`	`Object.isExtensible(proxy)`
`preventExtensions(target)`	`Object.preventExtensions(proxy)`
`getOwnPropertyDescriptor(target, key)`	`Object.getOwnPropertyDescriptor(proxy, key)`, `Object.keys(proxy)`
`ownKeys(target)`	`Object.keys(proxy)`, `Object.getOwnProperty(Names|Symbols)(proxy)`
`defineProperty(target, key, descriptor)`	`Object.defineProperty(proxy, key, descriptor)`
`apply(target, thisArg, args)`	`thisArg.proxy(...args)`, `proxy(...args)`, `proxy.apply(thisArg, args)`, `proxy.call(thisArg, ...args)`
`construct(target, args, newTarget)`	`new proxy(args)` или вызов с помощью `super`

В функциях `get` и `set` параметр `target` – это целевой объект прокси (здесь – `obj`), а `receiver` – объект, к свойству которого был произведен доступ, т. е. объект `proxy`, если только он не находится в цепочке прототипов другого объекта.

Теперь мы должны использовать прокси, а не оригинальный объект в любом коде, где хотим вести мониторинг.

Пусть кто-то изменил зарплату:

```
proxy.salary = 200000
```

Тогда порождается сообщение:

```
set salary to 200000
```

Неперехваченные операции передаются целевому объекту. В нашем примере вызов

```
delete proxy.salary
```

удалит поле `salary` из целевого объекта.

JavaScript API предлагает одну полезную реализацию прокси, которая позволяет передать проксированный объект коду, которому мы доверяем, а затем отозвать доступ, потому что мы не доверяем следующему далее коду.

Получить прокси можно следующим образом:

```
const target = . . .
const p = Proxy.revocable(target, {})
```

Функция `Proxy.revocable` возвращает объект, имеющий свойство `proxy` (проксированный объект), и метод `revoke`, который отзывает все права доступа к `proxy`.

Сначала передайте прокси коду, которому доверяете. Все операции будут производиться над целевым объектом.

После вызова

```
p.revoke() // использовать p.proxy больше нельзя
```

любая операция с `proxy` возбуждает исключение.

Вы обязаны предоставить обработчик для функций-перехватчиков. Если вас устраивает поведение по умолчанию, то предоставьте пустой объект. Пример нетривиального обработчика см. в упражнении 24.

11.13. Класс Reflect

Класс `Reflect` реализует все тринадцать перехватчиков, описанных в табл. 11.3. Мы можем вызывать соответствующие функции `Reflect`, вместо того чтобы реализовывать эти операции вручную:

```
const logHandler = {
  get(target, key, receiver) {
    console.log(`get ${key.toString()}`)
    return Reflect.get(target, key, receiver)
      // вместо return target[key]
  },
  set(target, key, value, receiver) {
    console.log(`set ${key.toString()}`)
    return Reflect.set(target, key, value, receiver)
      // вместо target[key] = value; return true
  }
}
```

Теперь предположим, что требуется протоколировать *все* допускающие перехват операции. Заметим, что код разных функций-обработчиков отличается только именем функции. Вместо того чтобы писать много почти идентичных обработчиков, можно написать второй прокси, который перехватывает акцессор чтения имени функции:

```
const getHandler = {
  get(target, trapKey, receiver) {
    return (...args) => {
      console.log(`Trapping ${trapKey}`)
```

```
      return Reflect[trapKey](...args);
    }
  }
}

const logEverythingHandler = new Proxy({}, getHandler)

const proxy = new Proxy(obj, logEverythingHandler)
```

Чтобы понять, что здесь происходит, рассмотрим конкретный сценарий.

1. Пользователь прокси устанавливает свойство:

   ```
   proxy.name = 'Fred'
   ```

2. Вызывается соответствующий метод `logEverythingHandler`:

   ```
   logEverythingHandler.set(obj, 'name', 'Fred', proxy)
   ```

3. Чтобы выполнить этот вызов, виртуальная машина должна найти метод `set` объекта `logEverythingHandler`.
4. Поскольку `logEverythingHandler` сам является прокси, то вызывается метод `get` его обработчика:

   ```
   getHandler.get({}, 'set', logEverythingHandler)
   ```

5. Этот вызов возвращает функцию

   ```
   (...args) => { console.log(`Trapping set`); return Reflect.set(...args) }
   ```

   в качестве значения `logEverythingHandler.set`.
6. Теперь может продолжить работу вызов функции, начатый на шаге 2. Эта функция вызывается с аргументами (`obj`, `'name'`, `'Fred'`, `proxy`).
7. На консоли печатается сообщение, после чего следует вызов

   ```
   Reflect.set(obj, 'name', 'Fred', proxy)
   ```

8. В результате этого вызова в `obj.name` записывается значение `'Fred'`.

Если вы хотите протоколировать аргументы перехватчиков (среди которых есть целевой объект и прокси), то нужно очень внимательно следить, чтобы не возникло бесконечной рекурсии. Например, можно хранить отображение известных объектов, которые печатаются по имени, а не вызывать метод `toString`, что привело бы к дополнительным вызовам перехватчиков.

```
const knownObjects = new WeakMap()

const stringify = x => {
  if (knownObjects.has(x))
    return knownObjects.get(x)
  else
    return JSON.stringify(x)
}

const logEverything = (name, obj) => {
```

```
  knownObjects.set(obj, name)
  const getHandler = {
    get(target, trapKey, receiver) {
      return (...args) => {
        console.log(`Trapping ${trapKey}(${args.map(stringify)})`)
        return Reflect[trapKey](...args);
      }
    }
  }

  const result = new Proxy(obj, new Proxy({}, getHandler))
  knownObjects.set(result, `proxy of ${name}`)
  return result
}
```

Теперь можно вызвать:

```
const fred = { name: 'Fred' }
const proxyOfFred = logEverything('fred', fred)
proxyOfFred.age = 42
```

В протоколе будут напечатаны следующие предложения:

```
Trapping set(fred,age,42,proxy of fred)
Trapping getOwnPropertyDescriptor(fred,age)
Trapping defineProperty(fred,"age",{"value":42,
  "writable":true,"enumerable":true,"configurable":true})
```

Класс `Reflect` предназначался для работы с прокси, но три его метода имеют самостоятельную ценность, поскольку немного удобнее классических аналогов.

1. `Reflect.deleteProperty` возвращает значение типа `boolean`, которое говорит, было ли удаление успешным. Оператор `delete` такой информации не дает.
2. `Reflect.defineProperty` возвращает значение типа `boolean`, которое говорит, было ли определение успешным. Функция `Object.defineProperty` в случае неудачи возбуждает исключение.
3. `Reflect.apply(f, thisArg, args)` гарантированно вызывает `Function.prototype.apply`, тогда как `f.apply(thisArg, args)` может этого не делать, если свойство `apply` было переопределено.

11.14. Инварианты прокси

Виртуальная машина проверяет, что ваши реализации операций прокси не возвращают явной ерунды. Например:

- функция `construct` должна возвращать объект;
- функция `getOwnPropertyDescriptor` должна возвращать объект-дескриптор или `undefined`;
- функция `getPrototypeOf` должна возвращать объект или `null`.

Кроме того, виртуальная машина проверяет согласованность операций прокси. Прокси должен учитывать некоторые аспекты своего целевого объекта, в том числе:

- не допускающие записи свойства целевого объекта;
- не допускающие конфигурирования свойства целевого объекта;
- нерасширяемость целевого объекта.

В спецификации ECMAScript описаны «инварианты», которые обязан сохранять прокси. Например, описание операции прокси `get` включает такое требование: «Значение свойства, сообщаемое `get`, должно совпадать со значением свойства соответствующего целевого объекта, если свойство целевого объекта является не допускающим ни записи, ни конфигурирования собственным свойством данных».

Аналогично, если свойство целевого объекта не допускает конфигурирования, то операция `has` не вправе скрывать его. Если целевой объект нерасширяемый, то операция `getPrototypeOf` должна возвращать истинный прототип, а `has` и `getOwnPropertyDescriptor` должны возвращать истинные свойства.

Эти инварианты имеют смысл, когда прокси дополняет существующий объект, не добавляя новых свойств. К сожалению, они заставляют нас лгать о свойствах, которые добавляет прокси. Рассмотрим массивоподобный объект для хранения диапазона значений, скажем целых чисел от 10 до 99. Хранить сами значения нет необходимости. Мы можем вычислять их динамически. В этом деле прокси особенно хороши. Приведенная ниже функция создает такой прокси диапазона:

```
const createRange = (start, end) => {
  const isIndex = key =>
    typeof key === 'string' && /^[0-9]+$/.test(key) && parseInt(key) < end - start

  return new Proxy({}, {
    get: (target, key, receiver) => {
      if (isIndex(key)) {
        return start + parseInt(key)
      } else {
        return Reflect.get(target, key, receiver)
      }
    }
  })
}
```

Перехватчик `get` порождает значения из диапазона по запросу:

```
const range = createRange(10, 100)
console.log(range[10]) // 20
```

Однако мы пока не можем перебрать ключи:

```
console.log(Object.keys(range)) // []
```

Это и неудивительно. Сначала нужно определить перехватчик `ownKeys`:

```
ownKeys: target => {
  const result = Reflect.ownKeys(target)
```

```
  for (let i = 0; i < end - start; i++)
    result.push(String(i))
  return result
}
```

К сожалению, даже после добавления перехватчика `ownKeys` в обработчик вызов `Object.keys(range)` возвращает пустой массив.

Чтобы исправить эту ошибку, мы должны предоставить дескрипторы для индексных свойств:

```
getOwnPropertyDescriptor: (target, key) => {
  if (isIndex(key)) {
    return {
      value: start + Number(key),
      writable: false,
      enumerable: true,
      configurable: true // не то, что нам действительно нужно
    }
  } else {
    return Reflect.getOwnPropertyDescriptor(target, key)
  }
}
```

Теперь `Object.keys` возвращает массив, содержащий значения от `'10'` до `'99'`. Однако в этой бочке меда есть капля дегтя. Индексные свойства обязаны быть конфигурируемыми. Иначе вмешиваются правила инвариантов. Нельзя сообщить о неконфигурируемом свойстве, которого нет в целевом объекте (а наш целевой объект пустой). На самом деле мы не хотим, чтобы индексные свойства были конфигурируемыми, но у нас связаны руки. Если требуется запретить удаление или переконфигурирование индексных свойств, то необходимо предоставить дополнительные перехватчики (см. упражнение 27).

Как видим, реализация динамических свойств в прокси – занятие не для слабых духом. Всюду, где возможно, размещайте свойства в целевом объекте прокси. Например, прокси диапазона должен иметь свойство `length` и метод `toString`. Ну так просто включите их в целевой объект и не обрабатывайте в перехватчиках (см. упражнение 28).

Упражнения

1. Почему метод `Symbol.toPrimitive` для класса `Percent` из раздела 11.2 неудовлетворителен? Попробуйте сложить и перемножить процентные значения. Почему нельзя исправить ошибку, так чтобы все работало и для арифметических операций с процентами, и для конкатенации строк?
2. Маска glob – это образец для сопоставления с именами файлов. В простейшей форме `*` сопоставляется с любой последовательностью знаков, отличных от разделителя пути `/`, а `?` – с одним знаком. Реализуйте класс `Glob`. Используя хорошо известные символы, разрешите применение маски glob в строковых методах `match`, `matchAll`, `replace`, `search` и `split`.

3. Из табл. 11.1 следует, что мы можем изменить поведение `x instanceof y`, гарантировав, что `y` имеет свойство, совпадающее с хорошо известным символом. Сделайте так, чтобы выражение `x instanceof Natural` проверяло, является ли `x` целым числом ⩾ 0, а `x instanceof Range(a, b)` – что `x` является целым числом в заданном диапазоне. Я не хочу сказать, что это хорошая идея, но интересен сам факт, что это можно сделать.
4. Определите класс `Person`, так чтобы для него и для всех его подклассов метод `toString` возвращал `[object Имя_класса]`.
5. Посмотрите, что возвращают следующие вызовы, и объясните результаты:

```
Object.getOwnPropertyDescriptors([1,2,3])
Object.getOwnPropertyDescriptors([1,2,3].constructor)
Object.getOwnPropertyDescriptors([1,2,3].prototype)
```

6. Предположим, что мы запечатали объект, вызвав функцию `Object.seal(obj)`. Попытка записи в несуществующее свойство в строгом режиме возбуждает исключение. Однако чтение несуществующего свойства не приводит к исключению. Напишите функцию `reallySeal` такую, что и чтение, и запись свойства, не существующего у возвращенного ей объекта, приводит к исключению. Указание: воспользуйтесь прокси.
7. Погуглите «JavaScript object clone» и почитайте несколько статей в блогах и ответов на сайте StackOverflow. Сколько из них корректно работают с разделяемым изменяемым состоянием и циклическими ссылками?
8. Напишите функцию `freezeCompletely`, которая замораживает сам объект и рекурсивно все значения его свойств. Обрабатывайте циклические зависимости.
9. С помощью функции `Object.getOwnPropertyDescriptors` найдите все свойства массива `[1, 2, 3]`, функции `Array` и объекта `Array.prototype`. Почему во всех трех случаях имеется свойство `length`?
10. Сконструируйте новый объект строки, вызвав `new String('Fred')`, и установите его прототип равным `Array.prototype`. Какие методы можно успешно применить к этому объекту? Начните с `map` и `reverse`.
11. Выражению `new.target`, с которым мы встретились в конце раздела 11.10, присваивается функция-конструктор, если объект сконструирован с помощью оператора `new`. Воспользовавшись этой особенностью, спроектируйте абстрактный класс `Person`, экземпляр которого нельзя создать с помощью `new`. Однако разрешите создание экземпляров конкретных подклассов, например `Employee`.
12. Как можно реализовать абстрактные классы с помощью свойства `constructor` прототипа, а не техники, описанной в предыдущем упражнении? Какой способ надежнее?
13. Выражение `new.target` равно `undefined`, если функция вызвана без `new`. Как можно проще распознать эту ситуацию в строгом режиме?
14. Исследуйте свойство `name` функций. Что в него записывается, когда в определении функции задано имя? А если имя не задано, но функция присвоена локальной переменной? А если это анонимная функция, переданная в качестве аргумента или возвращенная в качестве результата? А как насчет стрелочных выражений?

15. В разделе 11.11 мы видели, что метод `call` необходим для вызова метода из другого класса. Приведите аналогичный пример для `bind`.
16. В этом упражнении вы узнаете, как программисты на JavaScript были вынуждены реализовывать наследование до появления в языке ключевых слов `extends` и `super`. Дана функция-конструктор

```
function Employee(name, salary) {
 this.name = name
 this.salary = salary
}
```

Методы добавлены в прототип.

```
Employee.prototype.raiseSalary = function(percent) {
 this.salary *= 1 + percent / 100
}
```

Теперь реализуйте подкласс `Manager`, не используя ключевых слов `extends` и `super`. Используйте `Object.setPrototypeOf`, чтобы установить прототип `Manager.prototype`. В конструкторе `Manager` необходимо вызвать конструктор `Employee` от имени *существующего* объекта `this`, а не создавать новый. Используйте метод `bind`, описанный в разделе 11.11.

17. Пытаясь решить предыдущее упражнение, Фриц написал

```
Manager.prototype = Employee.prototype
```

вместо того чтобы использовать `Object.setPrototypeOf`. Каковы печальные последствия этого решения?

18. Как отмечено в конце раздела 11.8, `Array.prototype` действительно является массивом. Убедитесь в этом с помощью функции `Array.isArray`. Почему выражение `[] instanceof Array` равно `false`? Что происходит с массивами при добавлении элементов в массив `Array.prototype`?
19. Воспользуйтесь протоколирующим прокси из раздела 11.12 для мониторинга чтения и записи элементов массива. Что происходит при чтении или записи элемента? А свойства `length`? Что происходит, если проинспектировать объект прокси на консоли, набрав его имя?
20. Правда, неприятно, когда допускаешь опечатку в имени свойства или метода? С помощью прокси реализуйте автокоррекцию. Выбирайте ближайшее существующее имя. Вам понадобится какая-то мера близости строк, например количество общих символов или редакционное расстояние Левенштейна.
21. Поведение объектов, массивов или строк можно изменить, переопределив методы классов `Object`, `Array` или `String`. Реализуйте прокси, который запрещает такое переопределение.
22. Выражение вида `obj.prop1.prop2.prop3` возбуждает исключение, если любое из промежуточных свойств равно `null` или `undefined`. Давайте решим эту мелкую проблему с помощью прокси. Сначала определим безопасный объект, который возвращает самого себя при поиске любого свойства. Затем определим функцию такую, что `safe(obj)` является прокси для

obj, который возвращает безопасный объект, если запрошенное свойство имеет значение null или undefined. Дополнительный бонус тем, кто сумеет обобщить эту технику на вызовы методов, так чтобы выражение safe(obj).m1().m2().m3() не возбуждало исключения, когда любой из промежуточных методов возвращает null или undefined.

23. Создайте прокси, который поддерживает похожий на XPath синтаксис для поиска элементов в HTML- или XML-документе.

```
const root = makeRootProxy(document)
const firstItemInSecondList = root.html.body.ul[2].li[1]
```

24. Создайте отзываемый прокси, описанный в разделе 11.12, который делает все свойства доступными только для чтения, до тех пор, пока права доступа не будут отозваны окончательно.
25. В разделе 11.14 перехватчик getOwnPropertyDescriptor возвращает дескриптор индексных свойств, в котором атрибут configurable равен true. Что произойдет, если сделать его равным false?
26. Отладьте перехватчик ownKeys из раздела 11.14, протоколируя обращения к целевому объекту {}, для чего используйте метод logEverything из раздела 11.13. Также поместите вызов протоколирования в перехватчик getOwnPropertyDescriptor. Затем прочитайте раздел 9.5.11 стандарта ECMAScript 2020. Соответствует ли эта реализация алгоритму, описанному в стандарте?
27. Добавьте перехватчики в прокси диапазона из раздела 11.14, чтобы предотвратить удаление или модификацию индексных свойств. Добавьте также перехватчик has.
28. Добавьте свойство length и метод toString в прокси диапазона из раздела 11.14. Добавьте его в целевой объект прокси и не включайте специальной обработки в перехватчики. Установите подходящие атрибуты.
29. Прокси диапазона из раздела 11.14 создается путем вызова функции createRange. Используйте функцию-конструктор, так чтобы пользователь мог написать new Range(10, 100) и получить экземпляр прокси, который выглядит так, будто это экземпляр класса Range.
30. Продолжите предыдущее упражнение, так чтобы класс Range расширял Array. Не забудьте установить свойство Symbol.species, как описано в разделе 11.2.3.

Глава 12

Итераторы и генераторы

В этой короткой главе мы узнаем, как реализуются итераторы, которые можно использовать в цикле `for of` и в операторе расширения массива. Вы сможете работать с итераторами в собственном коде.

Реализация итератора – довольно трудоемкое занятие, но генераторы его существенно упрощают. Генератором называется функция, которая может отдавать несколько значений, приостанавливая работу после порождения каждого и возобновляя ее, когда запрашивается следующее значение. Генераторы также лежат в основе асинхронного программирования без обратных вызовов.

Все изложенное в этой главе считается материалом повышенного уровня.

12.1. Итерируемые значения

Пожалуй, чаще всего итерируемые значения в JavaScript встречаются в циклах `for of`. Например, массивы являются итерируемыми. Цикл

```
for (const element of [1, 2, 7, 9])
```

обходит все элементы данного массива. Строки – также итерируемые объекты, и цикл

```
for (const ch of 'Hello')
```

обходит кодовые точки данной строки.

Перечислим другие итерируемые значения:

- массивы и строки;
- множества и отображения;
- объекты, возвращенные методами `keys`, `values` и `entries` массивов, типизированных массивов, множеств и отображений (но не класса `Object`);
- структуры данных DOM, например возвращаемая вызовом метода `document.querySelectorAll('div')`.

В общем случае значение является итерируемым, если у него имеется метод с ключом `Symbol.iterator`, который возвращает объект-итератор:

```
const helloIter = 'Hello'[Symbol.iterator]()
```

У объекта-итератора имеется метод `next`, который возвращает объект, содержащий следующее значение, и признак завершения итераций:

```
helloIter.next() // возвращает { value: 'H', done: false }
helloIter.next() // возвращает { value: 'e', done: false }
...
helloIter.next() // возвращает { value: 'o', done: false }
helloIter.next() // возвращает { value: undefined, done: true }
```

В цикле

```
for (const v of iterable)
```

для получения объекта-итератора производится вызов `iterable[Symbol.iterator]()`. На каждой итерации цикла вызывается метод `next` этого объекта. Каждый вызов возвращает объект вида `{ value: ..., done: ... }`. Пока `done` равно `false`, переменной `v` присваивается свойство `value` объекта. Как только `done` станет равно `true`, цикл `for of` завершается.

Перечислим ситуации, когда в JavaScript используются итерируемые объекты:

- как уже было сказано, в цикле `for (const v of iterable)`;
- в операторе расширения массива: `[...iterable]`;
- совместно с деструктуризацией массива: `[first, second, third] = iterable`;
- совместно с функцией `Array.from(iterable)`;
- совместно с конструкторами множества и отображения: `new Set(iterable)`;
- совместно с директивой `yield*`, которую мы будем рассматривать ниже в этой главе;
- в любом месте, где программист пользуется итератором, сконструированным путем вызова функции, возвращенной в результате вызова `iterable[Symbol.iterable]()`.

12.2. Реализация итерируемого объекта

В этом разделе мы увидим, как создавать итерируемые объекты, которые могут встречаться в циклах `for of`, расширениях массивов и т. д.

Лучше всего сначала проработать конкретный пример. Давайте реализуем класс `Range`, в котором имеется итератор, возвращающий значения, принадлежащие заданному диапазону:

```
class Range {
  constructor(start, end) {
```

```
    this.start = start
    this.end = end
  }
  ...
}
```

Экземпляр `Range` можно будет использовать в цикле `for of` следующим образом:

```
for (const element of new Range(10, 20))
    console.log(element) // печатается 10 11 ... 19
```

У любого итерируемого объекта должен быть метод с именем `Symbol.iterator`. Поскольку имя метода – не строка, оно заключается в квадратные скобки:

```
class Range {
  ...
  [Symbol.iterator]() { ... }
}
```

Этот метод возвращает объект, имеющий метод `next`. Мы определим второй класс, порождающий такие объекты.

```
class RangeIterator {
  constructor(current, last) {
    this.current = current
    this.last = last
  }
  next() { ... }
}

class Range {
  ...
  [Symbol.iterator]() { return new RangeIterator(this.start, this.end) }
}
```

Метод `next` возвращает объекты вида { `value: ...`, `done: ...` }:

```
next() {
  ...
  if (...) {
    return { value: some value, done: false }
  } else {
    return { value: undefined, done: true }
  }
}
```

Если хотите, можете опускать `done: false` и `value: undefined`.

В нашем примере:

```
class RangeIterator {
  ...
  next() {
    if (this.current < this.last) {
```

```
      const result = { value: this.current }
      this.current++
      return result
    } else {
      return { done: true }
    }
  }
}
```

Благодаря явному определению двух классов становится очевидно, что метод `Symbol.iterator` возвращает экземпляр другого класса из метода `next`.

Альтернативно можно создавать объекты-итераторы на лету:

```
class Range {
  constructor(start, end) {
    this.start = start
    this.end = end
  }
  [Symbol.iterator]() {
    let current = this.start
    let last = this.end
    return {
      next() {
        if (current < last) {
          const result = { value: current }
          current++
          return result
        } else {
          return { done: true }
        }
      }
    }
  }
}
```

Метод `Symbol.iterator` возвращает объект, имеющий метод `next`, который возвращает объекты { `value: current` } и { `done: true` }.

Такая реализация более компактна, но читать ее труднее.

12.3. Закрываемые итераторы

Если объект-итератор обладает методом с именем `return` (именно так!), то он называется *закрываемым* (closeable). Метод `return` вызывается, когда итерирование завершается преждевременно. Предположим, к примеру, что `lines(filename)` осуществляет итерирование по строкам файла. Теперь рассмотрим функцию:

```
const find = (filename, target) => {
  for (line of lines(filename)) {
    if (line.contains(target)) {
```

```
      return line // вызывается iterator.return()
    }
  } // iterator.return() не вызывается
}
```

Метод `return` итератора вызывается, когда выход из цикла производится с помощью `return`, `throw`, `break` или `continue` с меткой. В примере выше метод `return` вызывается, когда строчка файла содержит строку `target`.

Если искомая строка ни разу не встретилась, то цикл `for of` завершается нормально и метод `return` не вызывается.

Если вы вручную вызываете метод `next` итератора и прекращаете работу с ним, до того как получен объект `done: true`, то должны сами вызвать `iterator.return()`.

Разумеется, после вызова `return` вызывать `next` уже нельзя.

Реализовывать закрываемый итератор не очень приятно, потому что логику закрытия нужно помещать в двух местах: в вызове `return` и в ветви метода `next`, которая обнаруживает, что больше значений нет.

Ниже приведена схематическая реализация функции, которая возвращает итерируемый объект для обхода строчек файла. В упражнении 6 вы должны будете восполнить недостающие детали.

```
const lines = filename => {
  const file = ... // открыть файл
  return {
    [Symbol.iterator]: () => ({
      next: () => {
        if (готово) {
          ... // закрыть файл
          return { done: true }
        } else {
          const line = ... // прочитать строчку
          return { value: line }
        }
      },
    ['return']: () => {
        ... // закрыть файл
        return { done: true } // должны вернуть объект
      }
    })
  }
}
```

12.4. Генераторы

В предыдущих разделах мы видели, как реализовать итератор, метод `next` которого порождает по одному значению за раз. Реализация может оказаться утомительной. Итератор должен запоминать состояние между последовательными обращениями к `next`. Даже случай простого диапазона и то не тривиален. К сожалению, нельзя просто воспользоваться циклом:

```
for (let i = start; i < end; i++)
  ...
```

Это не работает, потому что значения порождаются все сразу, а не по одному.

Однако в *генераторной функции* мы можем сделать именно то, что нужно:

```
function* rangeGenerator(start, end) {
  for (let i = start; i < end; i++)
    yield i
}
```

Ключевое слово `yield` порождает значение, но не означает выхода из функции. Функция приостанавливается после отдачи каждого значения. Когда понадобится следующее значение, функция продолжит выполнение с точки, следующей за предложением `yield`, и отдаст еще одно значение.

Знак `*` сообщает компилятору, что это генераторная функция. В отличие от обычной функции, которая может порождать только одно значение при возврате, генераторная функция порождает результат при каждом выполнении `yield`.

В момент вызова генераторной функции ее тело еще не начинает выполняться. Вместо этого мы получаем объект-итератор:

```
const rangeIter = rangeGenerator(10, 20)
```

Как и у любого итератора, у объекта `rangeIter` имеется метод `next`. При первом вызове `next` тело генераторной функции начинает работать и работает, пока не встретит предложение `yield`. Затем метод `next` возвращает объект { `value`: *`отданное значение`*, `done`: `false` }.

```
let nextResult = rangeIter.next() // { value: 10, done: false }
```

Начиная с этого момента при каждом вызове `next` выполнение генераторной функции возобновляется с последнего предложения `yield` и продолжается, пока снова не встретится `yield`.

```
nextResult = rangeIter.next() // { value: 11, done: false }
...
nextResult = rangeIter.next() // { value: 19, done: false }
```

Когда генераторная функция возвращает управление, метод `next` возвращает { `value`: *`возвращенное значение`*, `done`: `true` }, давая знать, что итерирование закончилось.

```
nextResult = rangeIter.next() // { value: undefined, done: true }
```

Если в какой-то момент код генераторной функции возбуждает исключение, то вызов `next` завершается с этим исключением.

> **Примечание.** В JavaScript предложение `yield` поверхностное – оно может встречаться только в самой генераторной функции, но не в функции, вызванной из нее.

Генераторная функция может быть именованной или анонимной:

```
function* myGenerator(...) { ... }
const myGenerator = function* (...) { ... }
```

Если свойство или метод объекта является генераторной функцией, оно должно быть помечено звездочкой:

```
const myObject = { * myGenerator(...) { ... }, ... }
  // Синтаксический сахар для myGenerator: function* (...) { ... }

class MyClass {
  * myGenerator(...) { ... }
  ...
}
```

Стрелочные функции не могут быть генераторами.

Вызов генераторной функции возможен в любом месте, где может находиться итерируемый объект, – в циклах `for of`, расширениях массивов и т. д.:

```
[...rangeGenerator(10, 15)] // массив [10, 11, 12, 13, 14]
```

12.5. Вложенное yield

Пусть требуется перебрать все элементы массива. Конечно, массив уже является итерируемым объектом, но все равно создадим итератор. Реализация тривиальна:

```
function* arrayGenerator(arr) {
  for (const element of arr)
    yield element
}
```

А что, если `arr` равно `[1, [2, 3, 4], 5]`, т. е. один элемент сам является массивом? В таком случае хотелось бы разгладить его при обходе и по очереди возвращать значения `1`, `2`, `3`, `4`, `5`. Первая попытка могла бы выглядеть так:

```
function* flatArrayGenerator(arr) {
  for (const element of arr)
    if (Array.isArray(element)) {
      arrayGenerator(element) // ошибка - не отдает никаких элементов
    } else {
      yield element
    }
}
```

Однако этот код не работает. Вызов

```
arrayGenerator(element)
```

не приводит к выполнению тела генераторной функции `arrayGenerator`. Мы просто получаем и отбрасываем итератор. Вызов

```
const result = [...flatArrayGenerator([1, [2, 3, 4], 5])]
```

присваивает result массив [1, 5].

Чтобы получить все значения генератора внутри генераторной функции, нужно воспользоваться предложением yield*:

```
function* flatArrayGenerator(arr) {
  for (const element of arr)
    if (Array.isArray(element)) {
      yield* arrayGenerator(element) // отдает сгенерированные элементы по одному
    } else {
      yield element
    }
}
```

Теперь вызов

```
const result = [...flatArrayGenerator([1, [2, 3, 4], 5])]
```

отдает разглаженный массив [1, 2, 3, 4, 5].

Однако если уровней вложенности несколько, то результат все равно получается неправильным:

```
flatArrayGenerator([1, [2, [3, 4], 5], 6]) // отдает значения 1, 2, [3, 4], 5, 6.
```

Исправить просто – нужно вызвать flatArrayGenerator рекурсивно:

```
function* flatArrayGenerator(arr) {
  for (const element of arr)
    if (Array.isArray(element)) {
      yield* flatArrayGenerator(element)
    } else {
      yield element
    }
}
```

Смысл этого примера в том, что yield* преодолевает ограничение генераторных функций в JavaScript. Каждое предложение yield должно находиться в теле самой генераторной функции. Оно не может встречаться в функции, вызываемой из генераторной. Предложение yield* разрешает ситуацию, когда одна генераторная функция вызывает другую, – при этом в общую последовательность вставляются значения, отданные вызванным генератором.

Предложение yield* также вставляет значения итерируемого объекта, отдавая по одному значению при каждом обращении к next. Это означает, что мы могли бы просто определить наш arrayGenerator как:

```
function* arrayGenerator(arr) {
  yield* arr
}
```

> **Примечание.** Генераторная функция может возвращать значения по завершении в дополнение к отдаче значений с помощью yield:

```
function* arrayGenerator(arr) {
  for (const element of arr)
    yield element
  return arr.length
}
```

Возвращаемое значение включается в результат последней итерации, когда свойство done равно true. При переборе отдаваемых значений возвращенное значение игнорируется. Но его можно получить как значение выражения yield* внутри еще одной генераторной функции:

```
function* elementsFollowedByLength(arr) {
  const len = yield* arrayGenerator(arr);
  yield len;
}
```

12.6. Генераторы как потребители

До сих пор мы использовали генераторы для порождения последовательности значений. Но генераторы могут также потреблять значения. При вызове метода next с аргументом он становится значением выражения yield:

```
function* sumGenerator() {
  let sum = 0
  while (true) {
    let nextValue = yield sum
    sum += nextValue
  }
}
```

Здесь значение выражения yield sum сохраняется в переменной nextValue и прибавляется к сумме. Имеет место двустороннее взаимодействие:

- генератор получает значения от стороны, вызвавшей метод next, и аккумулирует их;
- генератор отправляет текущую сумму стороне, вызвавшей метод next.

Предостережение. Необходимо начальное обращение к next, чтобы дойти до первого предложения yield. После этого можно вызывать next со значениями, которые будут потребляться генератором.

При вызове метода с именем return генератор останавливается, и последующие обращения к next дают { value: undefined, done: true }.

Ниже приведена полная последовательность обращений к итератору:

```
const accum = sumGenerator()
accum.next() // дойти до первого yield
let result = accum.next(3) // возвращает { value: 3, done: false }
result = accum.next(4) // возвращает { value: 7, done: false }
result = accum.next(5) // возвращает { value: 12, done: false }
accum.return() // останавливается и возвращает { value: undefined, done: true }
```

Вызов `throw(error)` от имени объекта итератора приводит к возбуждению исключения в ожидающем выражении `yield`. Если генераторная функция перехватывает исключение и доходит до предложения `yield` или `return`, то метод `throw` возвращает объект `{ value: ..., done: ... }`. Если генераторная функция завершается из-за неперехваченного исключения или потому, что было возбуждено другое исключение, то метод `throw` возбуждает это исключение.

Иными словами, `throw` ведет себя в точности как `next`, только выражение `yield` возбуждает исключение, а не отдает значение.

Для демонстрации `throw` рассмотрим такой вариант генератора суммы:

```
function* sumGenerator() {
  let sum = 0
  while (true) {
    try {
      let nextValue = yield sum
      sum += nextValue
    } catch {
      sum = 0
    }
  }
}
```

Вызов `throw` сбрасывает аккумулированную сумму:

```
const accum = sumGenerator()
accum.next() // дойти до первого yield
let result = accum.next(3)
result = accum.next(4)
result = accum.next(5)
accum.throw() // возвращается { value: 0, done; false }
```

Если вызвать `throw` до того, как достигнуто первое выражение `yield`, то генератор останавливается, при обращении к `throw` исключение возбуждается.

12.7. Генераторы и асинхронная обработка

После прочтения предыдущего раздела может возникнуть вопрос, зачем вообще может понадобиться генератор, аккумулирующий значения. Есть гораздо более простые способы вычислить сумму. Такие генераторы становятся куда более интересными в сочетании с асинхронным программированием.

Когда мы читаем данные из веб-страницы, они становятся доступны не мгновенно. В главе 9 мы видели, что в JavaScript имеется единственный поток выполнения. Если ждать, пока некое событие произойдет, то ничего другого программа делать не сможет. Поэтому веб-запросы выполняются асинхронно. Когда запрошенные данные станут доступны, будет вызвана заданная нами функция обратного вызова. Например, в следующем фрагменте

мы получаем случайное число, пользуясь классом XMLHttpRequest, встроенным в веб-браузеры (но не в Node.js):

```
const url = 'https://www.random.org/integers/?num=1&min=1&max=1000000000\
&col=1&base=10&format=plain&rnd=new'
const req = new XMLHttpRequest();
req.open('GET', url)
req.addEventListener('load', () => console.log(req.response)) // обратный вызов
req.send()
```

Поместим этот код в функцию, которая принимает в качестве параметра функцию-обработчик, вызываемую, когда будет получено случайное число:

```
const trueRandom = handler => {
  const url = 'https://www.random.org/integers/?num=1&min=1&max=1000000000\
&col=1&base=10&format=plain&rnd=new'
  const req = new XMLHttpRequest();
  req.open('GET', url)
  req.addEventListener('load', () => handler(parseInt(req.response)))
  req.send()
}
```

Теперь легко получить случайное число:

```
trueRandom(receivedValue => console.log(receivedValue))
```

Но допустим, что мы хотим сложить три таких числа. Тогда нужно сделать три вызова и вычислить сумму, когда будут готовы все ответы. Это занятие не для слабых духом:

```
trueRandom(first =>
    trueRandom(second =>
      trueRandom(third => console.log(first + second + third))))
```

Конечно, как мы видели в главе 9, в этой ситуации мы можем воспользоваться обещаниями и синтаксисом async/await. Но на самом деле обещания построены на основе генераторов. В этом разделе мы приведем краткий обзор того, как генераторы помогают организовать асинхронную обработку.

Воспользуемся генератором, чтобы создать иллюзию синхронных вызовов. Ниже мы опишем функцию nextTrueRandom, которая возвращает случайное целое число генератору. А вот сам генератор:

```
function* main() {
  const first = yield nextTrueRandom()
  const second = yield nextTrueRandom()
  const third = yield nextTrueRandom()
  console.log(first + second + third)
}
```

Запуск генератора возвращает итератор:

```
const iter = main()
```

Это тот итератор, в который мы будем загружать значения по мере их готовности:

```
const nextTrueRandom = () => {
  trueRandom(receivedValue => iter.next(receivedValue))
}
```

Осталось сделать одну вещь. Итерации необходимо начать:

```
iter.next() // запустить процесс
```

Теперь функция `main` начинает исполняться. Она вызывает `nextTrueRandom` и приостанавливается в выражении `yield`, пока кто-нибудь не вызовет метод `next` итератора.

Это обращение к `next` не произойдет, пока не станут доступны асинхронные данные. И вот тут-то генераторы становятся интересными. Они позволяют приостановить вычисление и возобновить его позже, когда окажется доступно значение. В конечном итоге значение приходит, и функция `nextTrueRandom` вызывает `iter.next(receivedValue)`. Это значение сохраняется в `first`.

Затем выполнение снова приостанавливается во втором выражении `yield` и т. д. В конце концов, мы получим все три значения и сможем вычислить их сумму.

В течение недолгого времени после включения генераторов в стандарт ES7 они считались решением, позволяющим избежать асинхронных обратных вызовов. Однако, как мы видели, такой подход интуитивно не очевиден. Гораздо проще использовать обещания и синтаксис `async/await`, описанный в главе 9. Генераторы, потребляющие значения, стали важным шагом на пути к обещаниям, но сами по себе нечасто используются прикладными программистами.

12.8. Асинхронные генераторы и итераторы

Генераторная функция отдает значения, которые можно получить с помощью итератора. При каждом вызове `iter.next()` генератор выполняется до следующего предложения `yield`, после чего приостанавливается.

Асинхронный генератор похож на генераторную функцию, но внутри его тела разрешается использовать оператор `await`. Концептуально асинхронный генератор порождает последовательность значений в будущем.

Для объявления асинхронного генератора используются одновременно ключевое слово `async` и звездочка `*`, обозначающая генераторную функцию:

```
async function* loadHanafudaImages(month) {
  for (let i = 1; i <= 4; i++) {
    const img = await loadImage(`hanafuda/${month}-${i}.png`)
    yield img
  }
}
```

При вызове асинхронного генератора мы получаем итератор. Но при вызове метода `next` этого итератора следующее значение может быть еще недоступно. Неизвестно даже, продолжается ли текущая итерация. Поэтому `next` возвращает обещание вернуть объект { `value: ..., done: ...` }.

Конечно, можно получить обещанные значения от итератора, но это утомительно (см. упражнение 16). Проще использовать специальную форму цикла `for` – `for await of`:

```
for await (const img of loadHanafudaImages(month)) {
  imgdiv.appendChild(img)
}
```

Цикл `for await of` должен находиться внутри асинхронной функции, поскольку вызывает оператор `await` для каждого сгенерированного обещания.

Если какое-то обещание будет отвергнуто, то цикл `for await of` возбудит исключение и итерирование завершится.

Цикл `for await of` работает с любым *асинхронным итерируемым объектом*, т. е. с объектом, обладающим свойством с ключом `Symbol.asyncIterator`, значением которого является функция, отдающая *асинхронный итератор*. У асинхронного итератора имеется метод `next`, возвращающий обещания вернуть объект вида { `value: ..., done: ...` }. Асинхронные генераторы – самый удобный механизм порождения асинхронных итерируемых объектов, но такой механизм можно реализовать и вручную (см. упражнение 17).

> **Предостережение.** Асинхронные итерируемые объекты *не являются* итерируемыми объектами. Они не работают в циклах `for of`, расширениях и совместно с деструктуризацией. Например, следующий код некорректен:
>
> ```
> const results = [...loadHanafudaImages(month)]
>   // Ошибка, это не массив обещаний
> for (const p of loadHanafudaImages(month)) p.then(imgdiv.appendChild(img))
>   // Ошибка, это не цикл по обещаниям
> ```
>
> **Примечание.** С другой стороны, цикл `for await of` может работать с обычными итерируемыми объектами. В этом случае он совпадает с циклом `for of`.

Ниже приведен пример асинхронного итерируемого объекта, который порождает диапазон чисел с задержкой между ними:

```
class TimedRange {
  constructor(start, end, delay) {
    this.start = start
    this.end = end
    this.delay = delay
  }

  async *[Symbol.asyncIterator]() {
    for (let current = this.start; current < this.end; current++) {
      yield await produceAfterDelay(current, this.delay)
    }
  }
}
```

Благодаря синтаксису `await` и `yield` реализация итераторной функции не вызывает затруднений. Просто нужно дождаться, когда станет доступно следующее значение, и отдать его.

Результаты можно потребить в цикле `for await of`:

```
let r = new TimedRange(1, 10, 1000)
for await (const e of r) console.log(e)
```

В заключение рассмотрим более реалистичный пример. Во многих API имеется параметр `page`, который позволяет последовательно выбирать страницы данных, например:

```
https://chroniclingamerica.loc.gov/search/titles/results/
  ?terms=michigan&format=json&page=5
```

Ниже показано, как разбить на страницы результаты такого запроса:

```
async function* loadResults(url) {
  let page = 0
  try {
    while (true) {
      page++
      const response = await fetch(`${url}&page=${page}`)
      yield await response.json()
    }
  } catch {
    // Конец итераций
  }
}
```

Вызывая генератор в цикле `for async of`, мы обойдем все ответы. Само по себе это не особенно интересно. Можно было бы реализовать такой обход в асинхронной функции без всякого генератора.

Однако этот генератор можно использовать как строительный блок для создания других полезных функций. Обычно разбиение на страницы используется, потому что автор приложения ожидает, что клиент остановится, найдя нужный ему результат. Вот как можно реализовать поиск, прекращающийся, как только функция обратного вызова вернула `true`:

```
const findResult = async (queryURL, callback) => {
  for await (const result of loadResults(queryURL)) {
    if (callback(result)) return result
  }
  return undefined
}
```

Обратите внимание на два момента. Во-первых, `findResult` – не генератор, а просто асинхронная функция. Если поместить трудную часть вычисления в асинхронный генератор, то результаты можно будет потребить с помощью любой асинхронной функции. Кроме того – и это очень важно, – получение страниц организовано лениво. Как только будет найдено соответствие,

функция `findResult` возвращает управление, отменяя генератор и прекращая выборку последующих страниц.

Упражнения

1. Напишите функцию, которая получает итерируемое значение и печатает каждый второй элемент.
2. Напишите функцию, которая получает итерируемое значение и возвращает другое итерируемое значение, которое отдает каждый второй элемент.
3. Реализуйте итерируемый объект, который отдает бесконечное число результатов бросания кости, т. е. случайных чисел от 1 до 6. Код должен занимать одну строчку:

```
const dieTosses = { ... }
```

4. Напишите функцию `dieTosses(n)`, которая возвращает итерируемый объект, отдающий `n` случайных целых чисел от 1 до 6.
5. Что неправильно в следующей реализации итератора `Range`?

```
class Range {
  constructor(start, end) {
    this.start = start
    this.end = end
  }
  [Symbol.iterator]() {
    let current = this.start
    return {
      next() {
        current++
        return current <= this.end ? { value: current - 1 } : { done: true }
      }
    }
  }
}
```

6. Завершите реализацию файлового итератора, начатую в разделе 12.3. Воспользуйтесь методами `openSync`, `readSync` и `closeSync` из модуля `fs`, входящего в состав Node.js (https://nodejs.org/api/fs.html). Не забудьте, что файл нужно закрывать в обеих функциях `next` и `return`. Чтобы избежать дублирования кода, можно вызывать `return` из `next`.
7. Измените функцию `arrayGenerator` из раздела 12.5, так чтобы для элементов массива, являющихся строками, каждый символ отдавался по отдельности.
8. Обобщите предыдущее упражнение – значения любого элемента массива, являющегося итерируемым объектом, должны отдаваться по отдельности.

9. Используя генератор, создайте итератор по дереву, который посещает узлы дерева по одному. Если вы знакомы с DOM API, посетите узлы документа DOM. В противном случае создайте свой класс дерева.
10. Воспользовавшись генератором и алгоритмом Хипа (https://en.wikipedia.org/wiki/Heap%27s_algorithm), создайте итератор, который отдает все перестановки массива. Например, для массива `[1, 2, 3]` итератор должен отдавать перестановки `[1, 2, 3]`, `[1, 3, 2]`, `[2, 3, 1]`, `[2, 1, 3]`, `[3, 1, 2]` и `[3, 2, 1]` (необязательно в таком порядке).
11. Как заставить метод `return` объекта-генератора возвращать значение? А нужно ли это?
12. В разделе 12.6 перечислено несколько вариантов поведения метода `throw`. Составьте таблицу, в которой будет описан каждый сценарий и ожидаемое поведение. Напишите короткие программы для демонстрации поведения в каждом случае.
13. Напишите функцию `trueRandomSum(n, handler)`, которая вычисляет сумму `n` случайных чисел и передает ее заданному обработчику. Используйте генератор, следуя идеям из раздела 12.6 «Генераторы как потребители».
14. Повторите предыдущее упражнение без использования генератора.
15. Рассмотрим следующую асинхронную функцию:

```
const putTwoImages = async (url1, url2, element) => {
  const img1 = await loadImage(url1)
  element.appendChild(img1)
  const img2 = await loadImage(url2)
  element.appendChild(img2)
  return element
}
```

А теперь рассмотрим следующую генераторную функцию, отдающую обещания:

```
function* putTwoImagesGen(url1, url2, element) {
  const img1 = yield loadImage(url1)
  element.appendChild(img1)
  const img2 = yield loadImage(url2)
  element.appendChild(img2)
  return element
}
```

По существу, это именно то преобразование, которому компилятор JavaScript подвергает любую асинхронную функцию. Теперь заполните пропуски `___` в функции `genToPromise`, которая принимает произвольный генератор, отдающий обещания, и преобразует его в объект `Promise`:

```
const genToPromise = gen => {
  const iter = gen()
  const nextPromise = arg => {
    const result = ___
    if (result.done) {
      return Promise.resolve(___)
    } else {
```

```
        return Promise.resolve(___).then(___)
      }
    }
    return nextPromise()
  }
```

16. Воспользуйтесь итератором, возвращенным генераторной функцией `loadHanafudaImages` из раздела 12.8, чтобы добавить все изображения в элемент DOM. Не используйте цикл `for await of`.
17. Реализуйте класс `TimedRange` из раздела 12.8, не пользуясь генераторной функцией. Создайте итератор, отдающий обещания, вручную.
18. Одно из возможных применений цикла `for await of` видится в сочетании с функцией `Promise.all`. Пусть имеется массив URL-адресов изображений. Преобразуем его в массив обещаний:

    ```
    const imgPromises = urls.map(loadImage)
    ```

    Запустим их параллельно, дождемся результирующего обещания и обойдем ответы. Какие из четырех показанных ниже циклов работают без ошибок? Какой следует использовать?

    ```
    for (const img of Promise.all(imgPromises)) element.appendChild(img)
    for await (const img of Promise.all(imgPromises)) element.appendChild(img)
    for (const img of await Promise.all(imgPromises)) element.appendChild(img)
    for await (const img of await Promise.all(imgPromises)) element.appendChild(img)
    ```

19. Какие из показанных ниже циклов работают без ошибок? Чем их поведение отличается от циклов из предыдущего упражнения?

    ```
    for (const p of urls.map(loadImage))
      p.then(img => element.appendChild(img))
    for (const p of urls.map(async url => await loadImage(url)))
      element.appendChild(await p)
    for await (const img of urls.map(url => await loadImage(url)))
      element.appendChild(img)
    for (const img of await urls.map(loadImage))
      element.appendChild(img)
    for await (const img of await urls.map(loadImage))
      element.appendChild(img)
    ```

20. В некоторых API (например, GitHub API, описанном в документе по адресу https://developer.github.com/v3/guides/traversing-with-pagination) результаты разбиения на страницы отдаются с помощью механизма, несколько отличающегося от описанного в разделе 12.8. Заголовок `Link` каждого ответа содержит URL для перехода к следующему результату. Получить его можно таким образом:

    ```
    let nextURL
      = response.headers.get('Link').match(/<(?<next>.*?)>; rel="next"/).groups.next;
    ```

    Адаптируйте генераторную функцию `loadResults` к этому механизму. Дополнительные бонусы – за раскрытие тайны регулярного выражения.

Глава 13

Введение в TypeScript

TypeScript представляет собой расширение JavaScript с добавлением проверки типов на этапе компиляции. Переменные и функции аннотируются ожидаемыми типами, а TypeScript выдает сообщение об ошибке, если программа нарушает правила типизации. Вообще говоря, это хорошо. Гораздо дешевле исправлять ошибки на стадии компиляции, чем отлаживать неправильно работающую программу. Кроме того, располагая информацией о типах, средства разработки могут лучше поддержать автозавершение и рефакторинг.

В этой главе содержится краткое введение в основные возможности TypeScript. Как и всюду в данной книге, я обращаю внимание прежде всего на современные средства, а унаследованные конструкции упоминаю лишь мельком. Цель главы – дать достаточно информации, чтобы вы могли сами решить, стоит ли использовать TypeScript поверх JavaScript.

Почему вообще кому-то может приглянуться TypeScript? В отличие от ECMAScript, развитие которого регулируется комитетом по стандартизации, включающим представителей многих компаний, TypeScript поддерживается только одним производителем – корпорацией Microsoft. В отличие от ECMAScript, для которого имеются документы, описывающие корректное поведение в мельчайших деталях, документация по TypeScript отрывочная и незавершенная. TypeScript – как и JavaScript – в каких-то отношениях непоследователен, что является еще одним потенциальным источником разочарования и путаницы. План выхода версий TypeScript отличается от плана для ECMAScript, т. е. мы получаем дополнительную подвижную часть. И наконец, в комплекте инструментов появляется еще одна программа, которая может работать неправильно.

Вы сами должны взвесить плюсы и минусы. В этой главе вы получите представление о TypeScript и сможете принимать решение обдуманно.

> **Совет.** Если, прочитав эту главу, вы придете к выводу, что статическая проверка типов вам интересна, но насчет TypeScript есть сомнения, ознакомьтесь с системой Flow (https://flow.org) – быть может, вам понравится ее система типов, синтаксис и инструментальные средства.

13.1. Аннотации типов

Рассмотрим функцию JavaScript, которая вычисляет среднее арифметическое двух чисел:

```
const average = (x, y) => (x + y) / 2
```

Что будет, если вызвать

```
const result = average('3', '4')
```

В этом случае строки `'3'` и `'4'` конкатенируются, получается строка `'34'`, которая затем преобразуется в число `34` и делится на 2, что дает `17`. Совершенно не то, что мы хотели.

В таких ситуациях JavaScript не сообщает об ошибках. Программа молча вычисляет неправильный результат и продолжает работать. Скорее всего, что-то плохое произойдет в другом месте.

В TypeScript мы аннотируем параметры:

```
const average = (x: number, y: number) => (x + y) / 2
```

Теперь ясно, что функция `average` должна вычислять среднее двух *чисел*. Если вызвать

```
const result = average('3', '4') // TypeScript: ошибка на этапе компиляции
```

то компилятор TypeScript сообщит об ошибке.

Именно в этом и состоит обещание TypeScript: мы включаем аннотации типов, а TypeScript обнаруживает ошибки типизации еще до запуска программы. Таким образом, мы проводим гораздо меньше времени в отладчике.

В этом примере процесс аннотирования очень прост. Но рассмотрим более сложный пример. Предположим, что аргумент может быть числом или массивом чисел. В TypeScript для описания такой ситуации используется *тип объединения*: `number | number[]`. В примере ниже мы хотим заменить целевое значение или несколько целевых значений другим значением:

```
const replace = (arr: number[], target: number | number[], replacement: number) => {
  for (let i = 0; i < arr.length; i++) {
    if (Array.isArray(target) && target.includes(arr[i])
        || !Array.isArray(target) && target === arr[i]) {
      arr[i] = replacement
    }
  }
}
```

Теперь TypeScript может проверить, что вызовы правильны:

```
const a = [11, 12, 13, 14, 15, 16]
replace(a, 13, 0) // OK
replace(a, [13, 14], 0) // OK
replace(a, 13, 14, 0) // Ошибка
```

Предостережение. TypeScript знает о типах методов из библиотеки JavaScript, но на момент написания книги онлайновая тестовая среда была сконфигурирована неправильно и не распознавала метод `includes` класса `Array`. Надеюсь, что к тому моменту, когда вы будете читать этот текст, ошибка будет исправлена. Если нет, замените `target.includes(arr[i])` на `target.indexOf(arr[i]) >= 0`.

Примечание. В примерах выше были использованы стрелочные функции. Аннотации работают точно так же при использовании ключевого слова `function`:

```
function average(x: number, y: number) { return (x + y) / 2 }
```

Чтобы эффективно использовать TypeScript, нужно знать, как выражаются типы вида «массив типа `T`» или «тип `T` или тип `U`» в синтаксисе TypeScript. В большинстве распространенных ситуаций это просто. Но описания типов могут быть довольно сложными, и иногда приходится вмешиваться в процесс проверки типов. Впрочем, так обстоит дело со всеми реальными системами. Нужно приложить усилия, чтобы потом пожинать их плоды – обнаружение ошибок на этапе компиляции.

13.2. Запуск TypeScript

Самый простой способ поэкспериментировать с TypeScript – тестовая среда по адресу https://www.typescriptlang.org/play. Просто введите свой код и выполните его. Если навести мышь на значение, то вы увидите его тип. Ошибки подчеркиваются извилистыми линиями (см. рис. 13.1).

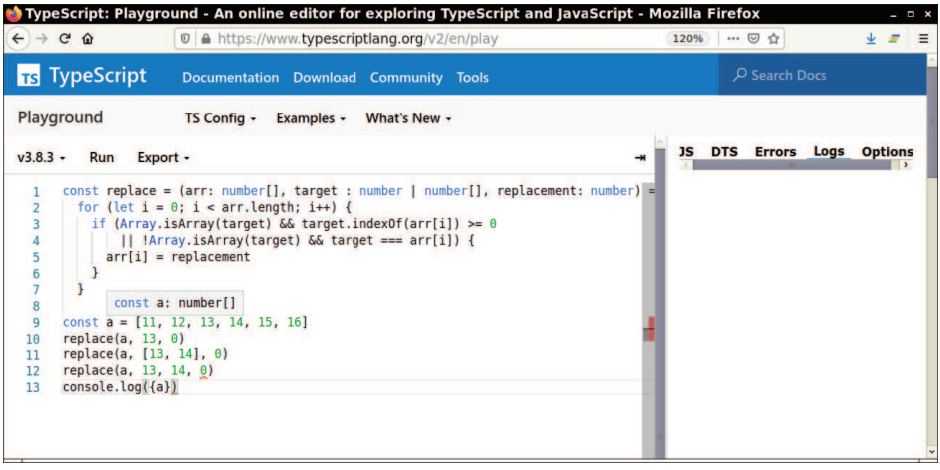

Рис. 13.1 ❖ Тестовая среда TypeScript

Visual Studio Code (https://code.visualstudio.com/) предлагает великолепную поддержку TypeScript, как, впрочем, и другие редакторы и интегрированные средства разработки.

Для работы с TypeScript из командной строки установите его с помощью менеджера пакетов npm командой

```
npm install -g typescript
```

В этой главе я всюду предполагаю, что TypeScript работает в *строгом режиме* и ориентирован на самую свежую версию ECMAScript. Как и обычный JavaScript, строгий режим TypeScript ставит вне закона «небрежную» унаследованную манеру программирования. Чтобы активировать все эти настройки, включите в каталог проекта файл tsconfig.json с таким содержимым:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "strict": true,
    "sourceMap": true
  },
  "filesGlob": [
    "*.ts"
  ]
}
```

Для компиляции TypeScript-файлов на JavaScript выполните команду

```
tsc
```

находясь в каталоге, содержащем TypeScript-файлы и файл tsconfig.json. Все TypeScript-файлы будут транслированы на JavaScript. Выполнить получившиеся файлы можно в Node.js.

Для запуска Node.js в режиме REPL введите команду

```
ts-node
```

находясь в каталоге, содержащем файл tsconfig.json, или команду

```
ts-node -O '{ "target": "es2020", "strict": true }'
```

если находитесь в любом другом каталоге.

13.3. Терминология, относящаяся к типам

Вернемся назад и поразмышляем о типах. Тип описывает множество значений, имеющих нечто общее. В TypeScript тип number состоит из всех значений, являющихся числами в JavaScript: обычных чисел вида 0, 3.141592653589793 и т. п., а также значений Infinity, -Infinity и NaN. Мы говорим, что все такие значения являются *экземплярами* типа number. Однако значение 'one' таковым не является.

Как мы уже видели, тип number[] – это массив чисел. Значение [0, 3.141592653589793, NaN] является экземпляром типа number[], а значение [0, 'one'] – нет.

Тип вида `number[]` называется *составным*. Мы можем создавать массивы любого типа: `number[]`, `string[]` и т. д. Объединение – еще один пример составного типа. Тип объединения

```
number | number[]
```

состоит из двух более простых типов: `number` и `number[]`.

С другой стороны, типы, не состоящие из более простых типов, называются *примитивными*. В TypeScript примитивными являются типы `number`, `string`, `boolean` и еще несколько, с которыми мы познакомимся в следующем разделе.

Составные типы могут быть весьма сложными. Чтобы их было проще воспринимать и повторно использовать, введено понятие *псевдонима типа*. Допустим, вы часто пишете функции, принимающие одиночное число или массив. Тогда просто определите псевдоним типа:

```
type Numbers = number | number[]
```

Псевдоним используется как сокращение типа:

```
const replace = (arr: number[], target: Numbers, replacement: number) => ...
```

> **Примечание.** Оператор `typeof` возвращает тип переменной или свойства. Этот тип можно использовать для объявления другой переменной того же типа:
>
> ```
> let values = [1, 7, 2, 9]
> let moreValues: typeof values = []
>   // typeof values - то же самое, что number[]
> let anotherElement: typeof values[0] = 42
>   // typeof values[0] - то же самое, что number
> ```

13.4. Примитивные типы

Любой примитивный тип JavaScript является также примитивным типом в TypeScript. Следовательно, примитивными типами в TypeScript являются `number`, `boolean`, `string`, `symbol`, `null` и `undefined`.

У типа `undefined` всего один экземпляр – значение `undefined`. Аналогично значение `null` – единственный экземпляр типа `null`. Вряд ли вы станете использовать эти типы сами по себе, но они очень полезны в составе объединений. Экземпляром типа

```
string | undefined
```

является либо строка, либо значение `undefined`.

Тип `void` может использоваться только как тип возвращаемого значения функции. Он означает, что функция не возвращает никакого значения (см. упражнение 2).

Тип `never` означает, что функция никогда не возвращает управления, потому что всегда возбуждает исключение. Поскольку обычно такие функции не пишут, маловероятно, что вам когда-нибудь доведется использовать тип

never в аннотации. В разделе 13.13.6 «Условные типы» описано еще одно применение типа never.

Тип unknown обозначает любое значение JavaScript. Любое значение можно привести к типу unknown, но значение типа unknown не совместимо ни с каким другим типом. Это имеет смысл для типов параметров очень общих функций (например, console.log) или когда необходим интерфейс с внешним JavaScript-кодом. Существует еще более слабый тип any. Разрешено любое преобразование в тип any или *из него*. Использование типа any следует свести к минимуму, потому что при этом отключается всякая проверка типов.

Литеральное значение обозначает тип с единственным экземпляром – таким значением. Например, строковый литерал 'Mon' – это тип в TypeScript. У этого типа имеется единственное значение – строка 'Mon'. Сам по себе такой тип не особенно полезен, но из них можно образовать тип объединения, например:

```
'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun'
```

Этот тип имеет семь экземпляров – названий дней недели.

Для таких типов обычно заводят псевдонимы:

```
type Weekday = 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun'
```

Теперь можно аннотировать переменную типом Weekday:

```
let w: Weekday = 'Mon' // OK
w = 'Mo' // ошибка
```

Такие типы, как Weekday, описывают конечное множество значений. Значения могут быть литералами любого типа:

```
type Falsish = false | 0 | 0n | null | undefined | '' | []
```

> **Примечание.** Если вам нужны константы с понятными именами, то TypeScript позволяет определить тип перечисления, например:
>
> ```
> enum Weekday { MON, TUE, WED, THU, FRI, SAT, SUN }
> ```
>
> К этим константам можно обращаться по именам Weekday.MON, Weekday.TUE и т. д. Это синонимы чисел 0, 1, 2, 3, 4, 5, 6. Можно также присваивать значения символическим константам:
>
> ```
> enum Color { RED = 4, GREEN = 2, BLUE = 1 }
> ```
>
> Строковые константы тоже допустимы:
>
> ```
> enum Quarter { Q1 = 'Winter', Q2 = 'Spring', Q3 = 'Summer', Q4 = 'Fall' }
> ```

13.5. Составные типы

TypeScript предлагает несколько способов построения сложных типов из более простых. Все они описаны в этом разделе.

Для любого типа существует тип массива:

```
number[] // массив number
string[] // массив string
number[][] // массив number[]
```

Эти типы описывают массивы, все элементы которых имеют одинаковый тип. Например, массив `number[]` может содержать только числа, но не смесь чисел и строк.

Разумеется, программисты на JavaScript часто используют массивы смешанных типов, например `[404, 'not found']`. В TypeScript такой массив описывается как экземпляр *типа кортежа* `[number, string]`. Тип кортежа – это список типов, заключенный в квадратные скобки. Он обозначает массивы фиксированной длины, элементы которых имеют заданные типы. В нашем примере значение `[404, 'not found']` является экземпляром типа кортежа `[number, string]`, а значения `['not found', 404]` или [404, `'error', 'not found']` – нет.

> **Примечание.** Тип массива, который начинается строкой и числом, а потом может иметь элементы других типов, обозначается
>
> `[string, number, ...unknown[]]`

Если тип кортежа описывает массивы, то *тип объекта* определяет имена и типы свойств объекта, например:

```
{ x: number, y: number }
```

Чтобы это объявление было проще использовать, можно воспользоваться псевдонимом типа:

```
type Point = { x: number, y: number }
```

Теперь можно определять функции, параметрами которых являются экземпляры типа `Point`:

```
const distanceFromOrigin = (p: Point) => Math.sqrt(Math.pow(p.x, 2) + Math.pow(p.y, 2))
```

Тип функции описывает типы параметров и возвращаемого значения функции. Например,

```
(arg1: number, arg2: number) => number
```

– это тип функций с двумя параметрами типа `number`, которые возвращают значение типа `number`.

Экземпляром такого типа является функция `Math.pow`, а `Math.sqrt` не является, потому что имеет всего один параметр.

> **Примечание.** В типе функций необходимо указывать не только типы параметров, но и имена, как показано в примере выше для параметров `arg1` и `arg2`. Эти имена обычно игнорируются, но есть одно исключение. Метод характеризуется тем, что первый параметр называется `this` – см. раздел 13.8.2. Во всех остальных случаях я буду использовать в типе функции имена `arg1`, `arg2` и т. д., чтобы сразу было видно, что это тип, а не настоящая функция. Параметр «прочие» я буду называть `rest`.

Мы уже встречались с типами объединения. Значениями типа *T* | *U* являются экземпляры *T* или *U*. Например, экземпляром типа

```
number | string
```

является либо число, либо строка, а тип

```
(number | string)[]
```

описывает массивы, элементами которых являются числа или строки.

Экземплярами *типа пересечения T* & *U* являются значения, одновременно удовлетворяющие требованиям *T* и *U*, например:

```
Point & { color: string }
```

Чтобы называться экземпляром этого типа, объект должен иметь числовые свойства `x` и `y` (это требования типа `Point`), а также строковое свойство `color`.

13.6. Выведение типа

Рассмотрим обращение к нашей функции `average`:

```
const average = (x: number, y: number) => (x + y) / 2
...
const a = 3
const b = 4
let result = average(a, b)
```

Только параметры функции нуждаются в аннотациях типов. Типы прочих переменных *выводятся*. Глядя на инициализацию, TypeScript может сказать, что `a` и `b` должны иметь тип `number`. Из анализа кода функции `average` TypeScript делает вывод, что возвращаемое значение тоже должно иметь тип `number`, а значит, таков тип `result`.

Вообще говоря, механизм выведения типов работает хорошо, но иногда нужно помочь TypeScript.

Начального значения переменной может быть недостаточно, чтобы определить тип, который имел в виду программист. Например, предположим, что объявляется тип для кодов ошибок:

```
type ErrorCode = [number, string]
```

Теперь мы хотим объявить переменную этого типа. Следующего объявления недостаточно:

```
let code = [404, 'not found']
```

TypeScript выводит из правой части тип `(number | string)[]`: массив произвольной длины, каждый элемент которого может быть числом или строкой. Это более общий тип, чем `ErrorCode`.

Совет. Чтобы узнать выведенный тип, воспользуйтесь средой разработки, которая отображает информацию о типах. На рис. 13.2 показано, как выведенные типы отображаются в Visual Studio Code.

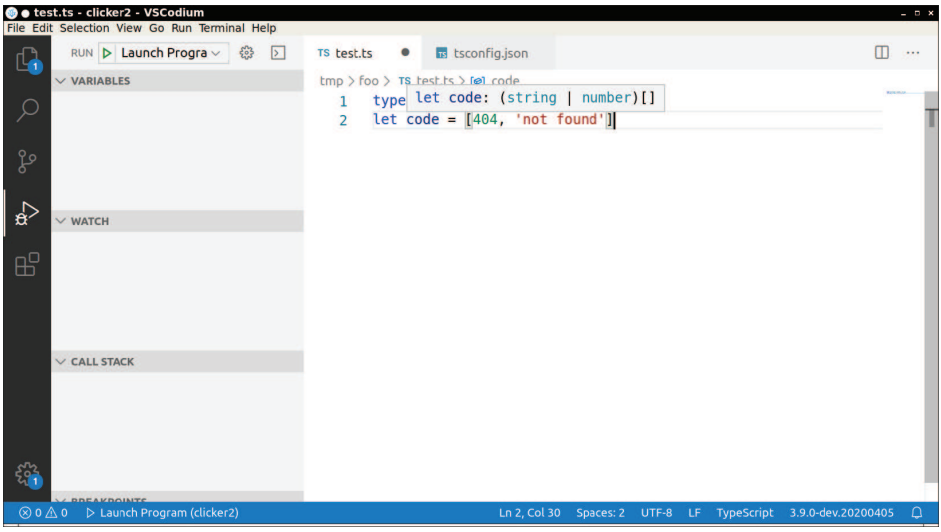

Рис. 13.2 ❖ Информация о типе в Visual Studio Code

Чтобы исправить ситуацию, нужно снабдить переменную аннотацией типа:

```
let code: ErrorCode = [404, 'not found']
```

Такая же проблема возникает, когда функция возвращает значение, тип которого не удается определить однозначно, например:

```
const root = (x: number) => {
  if (x >= 0) return Math.sqrt(x)
  else return [404, 'not found']
}
```

Будет выведен тип возвращаемого значения `number | (number | string)[]`. Если вам нужен тип `number | ErrorCode`, поставьте аннотацию типа после списка параметров:

```
const root = (x: number): number | ErrorCode => {
  if (x >= 0) return Math.sqrt(x)
  else return [404, 'not found']
}
```

А вот как выглядит объявление той же функции с ключевым словом `function`:

```
function root(x: number): number | ErrorCode {
  if (x >= 0) return Math.sqrt(x)
```

```
  else return [404, 'not found']
}
```

Аннотация типа нужна и тогда, когда переменная инициализируется значением `undefined`:

```
let result = undefined
```

Без аннотации TypeScript выведет тип `any`. (Было бы бессмысленно выводить тип `undefined` – тогда переменную нельзя было бы изменить.) Поэтому нужный тип следует задать явно:

```
let result: number | undefined = undefined
```

Впоследствии в `result` можно будет сохранить число, но не строку:

```
result = 3 // OK
result = '3' // ошибка
```

Иногда мы знаем о типе выражения больше, чем может вывести TypeScript. Например, допустим, что мы получили JSON-объект и знаем его тип. Тогда можно использовать *утверждение типа*:

```
let target = JSON.parse(response) as Point
```

Утверждение типа аналогично приведению в Java или C#, но если значение не соответствует целевому типу, то исключение не возбуждается.

При обработке типов объединения TypeScript следует потоку управления, чтобы гарантировать правильность типа в каждой ветви. Рассмотрим пример:

```
const less = (x: number | number[] | string | Date | null) => {
  if (typeof x === 'number')
    return x - 1;
  else if (Array.isArray(x))
    return x.splice(0, 1)
  else if (x instanceof Date)
    return new Date(x.getTime() - 1000)
  else if (x === null)
    return x
  else
    return x.substring(1)
}
```

TypeScript понимает смысл операторов `typeof`, `instanceof` и `in`, функции `Array.isArray` и проверок на `null` и `undefined`. Поэтому в первых четырех ветвях для переменной `x` выводятся типы `number`, `number[]`, `Date` и `null`. В пятой ветви остается только вариант `string`, поэтому TypeScript разрешает вызвать метод `substring`.

Но иногда такое выведение не работает. Рассмотрим пример:

```
const more = (values: number[] | string[]) => {
  if (array.length > 0 &&
    typeof x[0] === 'number') // ошибка - недопустимый охранник типа
```

```
    return values.map(x => x + 1)
  else
    return values.map(x => x + x)
}
```

TypeScript не может проанализировать это условие, оно для него слишком сложное. В такой ситуации можно предоставить специальную *функцию охраны типа*. Ее специальная роль подчеркивается типом возвращаемого значения:

```
const isNumberArray = (array: unknown[]): array is number[] =>
  array.length > 0 && typeof array[0] === 'number'
```

Тип возвращаемого значения `array is number[]` означает, что эта функция возвращает булево значение и может быть применена для проверки того, что аргумент `array` имеет тип `number[]`. Вот как эта функция используется:

```
const more = (values: number[] | string[]) => {
  if (isNumberArray(values))
    return values.map(x => x + 1)
  else
    return values.map(x => x + x)
}
```

А вот как тот же охранник типа выглядит, когда используется ключевое слово `function`:

```
function isNumberArray(array: unknown[]): array is number[] {
return array.length > 0 && typeof array[0] === 'number'
}
```

13.7. Подтипы

Некоторые типы, например `number` и `string`, никак не связаны между собой. Переменная типа `number` не может содержать значение типа `string`, и наоборот. Но есть и взаимосвязанные типы. Например, переменная типа `number | string` *может* содержать значение типа `number`.

Говорят, что `number` является подтипом `number | string`, а `number | string` – супертипом `number` и `string`. Подтип накладывает больше ограничений, чем его супертипы. В переменной, принадлежащей супертипу, можно хранить значения подтипа, но не наоборот.

В следующих разделах мы рассмотрим связь «является подтипом» более подробно.

13.7.1. Правило подстановки

Снова рассмотрим тип

```
type Point = { x: number, y: number }
```

Объект { x: 3, y: 4 }, очевидно, является экземпляром Point. А что можно сказать об объекте

```
const bluePoint = { x: 3, y: 4, color: 'blue' }
```

Он тоже является экземпляром Point? Ведь у него же есть свойства x и y, значениями которых являются числа.

В TypeScript ответ отрицательный. Объект bluePoint является экземпляром типа

```
{ x: number, y: number, color: string }
```

Для удобства поименуем этот тип:

```
type ColoredPoint = { x: number, y: number, color: string }
```

Тип ColoredPoint является подтипом Point, а Point – супертипом ColoredPoint. Подтип налагает все ограничения супертипа и добавляет еще какие-то.

Всюду, где ожидается значение некоторого типа, можно подставить экземпляр его подтипа. Иногда это называют *правилом подстановки*.

Например, ниже мы передаем объект ColoredPoint функции с параметром типа Point:

```
const distanceFromOrigin = (p: Point) => Math.sqrt(Math.pow(p.x, 2) + Math.pow(p.y, 2))
const result = distanceFromOrigin(bluePoint) // OK
```

Функция distanceFromOrigin ожидает получить Point, но готова принять и ColoredPoint. А почему бы и нет? Этой функции нужен доступ к числовым свойствам x и y, а они присутствуют.

> **Примечание.** Как мы только что видели, тип переменной не обязан совпадать с типом значения, на которое она ссылается. В примере выше параметр p имеет тип Point, а значение, на которое он ссылается, – тип ColoredPoint. Имея переменную некоторого типа, мы можем быть уверены, что значение, на которое она ссылается, принадлежит либо этому типу, либо его *подтипу*.

В TypeScript из правила подстановки есть одно исключение. Нельзя подставлять *объектный литерал* подтипа. Вызов

```
const result = distanceFromOrigin({ x: 3, y: 4, color: 'blue' }) // ошибка
```

не компилируется. Это называется *проверкой лишних свойств*.

Такая же проверка выполняется, когда объектный литерал присваивается типизированной переменной:

```
let p: Point = { x: 3, y: 4 }
p = { x: 0, y: 0, color: 'red' } // ошибка - добавлено свойство blue
```

С обоснованием этой проверки мы познакомимся в следующем разделе.

Обойти проверку лишних свойств нетрудно. Просто нужно завести еще одну переменную:

```
const redOrigin = { x: 0, y: 0, color: 'red' }
p = redOrigin // OK - в p может храниться значение подтипа
```

13.7.2. Факультативные и лишние свойства

Имея объект типа `Point`, мы можем читать только свойства `x` и `y`, поскольку нет никакой гарантии, что существуют еще какие-то свойства.

```
let p: Point = . . .
console.log(p.color) // ошибка - нет такого свойства
```

Это разумно. Ведь именно для таких проверок система типов и предназначена.

А как насчет записи свойства?

```
p.color = 'blue' // ошибка - нет такого свойства
```

С точки зрения теории типов это действие безопасно. Переменная `p` попрежнему ссылалась бы на значение, принадлежащее подтипу `Point`. Но TypeScript запрещает установку «лишних свойств».

Если вы хотите, чтобы свойства присутствовали в некоторых, но не во всех объектах типа, то следует использовать *факультативные свойства*. Свойство, помеченное знаком `?`, разрешено, но не обязательно. Например:

```
type MaybeColoredPoint = {
  x: number,
  y: number,
  color?: string
}
```

Теперь следующие предложения компилируются без ошибок:

```
let p: MaybeColoredPoint = { x: 0, y: 0 } // OK - свойство color факультативное
p.color = 'red' // OK - можно устанавливать факультативное свойство
p = { x: 3, y: 4, color: 'blue' } // OK - можно использовать литерал с
                                  // факультативным свойством
```

Цель проверки лишних свойств – обнаруживать опечатки в именах факультативных свойств. Рассмотрим функцию, которая рисует точку:

```
const plot = (p: MaybeColoredPoint) => . . .
```

Следующий вызов не компилируется:

```
const result = plot({ x: 3, y: 4, colour: 'blue' })
  // Ошибка - лишнее свойство colour
```

Обратите внимание на британское написание слова `colour`. В классе `MaybeColoredPoint` нет свойства `colour`, и TypeScript отлавливает эту ошибку. Если бы компилятор следовал правилу подстановки без проверки лишних свойств, то эта функция нарисовала бы точку без цвета (свойства `color`).

13.7.3. Вариантность типов массива и объекта

Верно ли, что массив цветных точек более специализирован, чем массив точек? Вроде бы да, какие могут быть сомнения? Действительно, в TypeScript тип `ColoredPoint[]` является подтипом `Point[]`. Вообще, если *S* – подтип *T*, то тип массива *S*[] – подтип *T*[]. Мы говорим, что в TypeScript массивы *ковариантны*, поскольку тип массива изменяется в том же направлении, что и типы его элементов.

Но на самом деле эта связь *некорректна*. На TypeScript можно написать программы, которые компилируются без ошибок, но выдают ошибки во время выполнения. Рассмотрим пример.

```
const coloredPoints: ColoredPoint[] = [{ x: 3, y: 4, color: 'blue' },
                                       { x: 0, y: 0, color: 'red' }]
const points: Point[] = coloredPoints // OK - точка может хранить значение подтипа
```

Мы можем добавить простую точку `Point` посредством переменной `points`:

```
points.push({ x: 4, y: 3 }) // OK - можно добавить Point в Point[]
```

Но `coloredPoints` и `points` *указывают на один и тот же массив*. При чтении добавленной точки через переменную `coloredPoints` возникает ошибка:

```
console.log(coloredPoints[2].color.length)
  // ошибка - невозможно прочитать свойство 'length' значения undefined
```

Значение `coloredPoints[2].color` равно `undefined`, чего не должно быть для `ColoredPoint`. В системе типов образовалась слепая зона.

Это осознанный выбор проектировщиков языка. Теоретически ковариантными должны быть только неизменяемые массивы, а изменяемые должны быть *инвариантными*. То есть не должно быть связи «является подтипом» между изменяемыми массивами разных типов. Однако с инвариантными массивами было бы неудобно работать. В этом случае в TypeScript так же, как в Java и C#, принято решение пожертвовать полной безопасностью типов ради удобства.

Ковариантность имеет место также для типов объектов. Чтобы определить, является ли один объект подтипом другого, мы смотрим на соответственные свойства. Рассмотрим два типа с одним общим свойством:

```
type Colored = { color: string }
type MaybeColored = { color: string | undefined }
```

В этом случае `string` является подтипом `string | undefined`, поэтому `Colored` – подтип `MaybeColored`.

Вообще, если *S* – подтип *T*, то тип объекта { *p*: *S* } – подтип { *p*: *T* }. Если свойств несколько, то все они должны изменяться в одном направлении.

Как и с массивами, ковариантность для объектов некорректна – см. упражнение 11.

В этом разделе мы видели, как типы объекта и массива зависят от типов компонентов. О вариантности типов функций см. раздел 13.12.3, а о вариантности в общем случае – раздел 13.13.5.

13.8. Классы

В следующих разделах описываются классы в TypeScript. Сначала мы рассмотрим синтаксические различия между классами в JavaScript и TypeScript, а затем покажем, как классы соотносятся с типами.

13.8.1. Объявление классов

Синтаксис классов в TypeScript похож на JavaScript. Конечно, мы снабжаем параметры конструктора и методов аннотациями типов. Также необходимо указать типы полей экземпляра. Вот один из способов перечислить поля с аннотациями типов:

```
class Point {
  x: number
  y: number

  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }

  distance(other: Point) {
    return Math.sqrt(Math.pow(this.x - other.x, 2) + Math.pow(this.y - other.y, 2))
  }

  toString() { return `(${this.x}, ${this.y})` }

  static origin = new Point(0, 0)
}
```

Вместо этого можно задать начальные значения, по которым TypeScript выведет тип:

```
class Point {
  x = 0
  y = 0
  ...
}
```

> **Примечание.** Этот синтаксис соответствует синтаксису полей в JavaScript, который находится на третьей стадии рассмотрения.

Поля экземпляра можно сделать закрытыми. TypeScript уже поддерживает тот синтаксис для описания закрытых средств, который в JavaScript пока находится на третьей стадии рассмотрения.

```
class Point {
  #x: number
  #y: number

  constructor(x: number, y: number) {
```

```
    this.#x = x
    this.#y = y
  }

  distance(other: Point) {
    return Math.sqrt(Math.pow(this.#x - other.#x, 2) + Math.pow(this.#y - other.#y, 2))
  }

  toString() { return `(${this.#x}, ${this.#y})` }

  static origin = new Point(0, 0)
}
```

Примечание. TypeScript поддерживает также модификаторы `private` и `protected` для полей и методов экземпляра. Эти модификаторы работают так же, как в Java и C++. Они остались от тех времен, когда в JavaScript не было синтаксиса для закрытых переменных и методов. В этой главе я не стану их обсуждать.

Примечание. Объявлять поля экземпляра можно с ключевым словом `readonly`:

```
class Point {
  readonly x: number
  readonly y: number
  ...
}
```

Так объявленное свойство нельзя изменять после начального присваивания.

```
const p = new Point(3, 4)
p.x = 0 // ошибка – нельзя изменять readonly-свойство
```

Заметим, что `readonly` относится к *свойствам*, тогда как `const` – к переменным.

13.8.2. Тип экземпляра класса

Экземпляры класса принадлежат типу TypeScript, содержащему все открытые свойства и методы. Например, рассмотрим класс `Point` с открытыми полями, описанный в предыдущих разделах. Его экземпляры имеют тип

```
{
  x: number,
  y: number,
  distance: (this: Point, arg1: Point) => number
  toString: (this: Point) => string
}
```

Заметим, что конструктор и статические члены не являются частями типа экземпляра.

Уточнить, что функция является методом, можно, назвав первый параметр `this`, как в примере выше. Альтернативно можно использовать следующую компактную нотацию:

```
{
  x: number,
  y: number,
```

```
  distance(arg1: Point): number
  toString(): string
}
```

Акцессоры чтения и записи в классах становятся свойствами в типах TypeScript. Например, если определить

```
get x() { return this.#x }
set x(x: number) { this.#x = x }
get y() { return this.#y }
set y(y: number) { this.#y = y }
```

в классе `Point` с закрытыми полями экземпляра из предыдущего раздела, то в типе TypeScript появятся свойства `x` и `y` типа `number`.

Если предоставить только акцессор чтения, то свойство будет доступно лишь для чтения (`readonly`).

> **Предостережение.** Если предоставить только акцессор записи, то чтение свойства разрешено, но возвращается `undefined`.

13.8.3. Статический тип класса

В предыдущем разделе отмечено, что конструктор и статические члены не являются частью типа экземпляра класса. Они принадлежат статическому типу.

Статический тип нашего класса `Point` имеет вид

```
{
  new (x: number, y: number): Point
  origin: Point
}
```

Синтаксис задания конструктора почти такой же, как для метода, но вместо имени метода используется ключевое слово `new`.

Обычно можно не задумываться о статическом типе (см., однако, раздел 13.13.4 «Стирание»). Тем не менее он часто становится причиной путаницы. Рассмотрим такой фрагмент кода:

```
const a = new Point(3, 4)
const b: typeof a = new Point(0, 0) // OK
const ctor: typeof Point = new Point(0, 0) // ошибка
```

Поскольку `a` – экземпляр `Point`, `typeof a` – экземпляр типа класса `Point`. Но что такое `typeof Point`? Здесь `Point` – функция-конструктор. В конце концов, класс в JavaScript – это не что иное, как функция-конструктор. Ее типом является статический тип класса. Можно инициализировать переменную `ctor` следующим образом:

```
const ctor: typeof Point = Point
```

Теперь можно вызвать `new ctor(3, 4)` или обратиться к свойству `ctor.origin`.

13.9. Структурная типизация

В системе типов TypeScript используется *структурная типизация*. Два типа считаются одинаковыми, если они имеют одинаковую структуру. Например,

```
type ErrorCode = [number, string]
```

и

```
type LineItem = [number, string]
```

– это один и тот же тип. Имена типов несущественны. Значения одинаковых типов можно присваивать друг другу:

```
let code: ErrorCode = [404, 'Not found']
let items: LineItem[] = [[2, 'Blackwell Toaster']]
items[1] = code
```

Выглядит потенциально опасно, но, безусловно, ничем не хуже того, что программисты каждодневно делают в программах на обычном JavaScript. К тому же на практике крайне маловероятно, что у двух типов будет в точности одинаковая структура. В нашем примере мы скорее пришли бы к таким типам:

```
type ErrorCode = { code: number, description: string }
type LineItem = { quantity: number, description: string }
```

Они различаются, потому что имена свойств не совпадают.

Структурная типизация принципиально отлична от «номинальных» систем типов в Java, C# и C++, где имена типов имеют значение. Но в JavaScript важны возможности объекта, а не имя его типа.

Для иллюстрации этого различия рассмотрим такую JavaScript-функцию:

```
const act = x => { x.walk(); x.quack(); }
```

Очевидно, что в JavaScript эта функция работает с любым `x`, имеющим методы `walk` и `quack`.

В TypeScript это поведение можно точно отразить в описании типа:

```
const act = (x: { walk(): void, quack(): void }) => { x.walk(); x.quack(); }
```

Мы можем завести класс `Duck`, предоставляющий эти методы:

```
class Duck {
  constructor(...) { ... }
  walk(): void { ... }
  quack(): void { ... }
}
```

Отлично. Мы можем передать экземпляр `Duck` функции `act`:

```
const donald = new Duck(...)
act(donald)
```

Но предположим, что у нас есть еще один объект – он не является экземпляром этого класса, но все равно обладает методами `walk` и `quack`:

```
const daffy = { walk: function () { ... }, quack: function () { ... } };
```

Его с тем же успехом можно передать функции `act`. Этот феномен называется «утиной типизацией» с отсылкой к пословице «если нечто ходит, как утка, и крякает как утка, то это утка и есть».

Структурная типизация в TypeScript формализует этот подход. Пользуясь структурой типа, TypeScript может на этапе компиляции проверить, что каждое значение обладает необходимыми возможностями. Имя типа при этом не играет никакой роли.

13.10. Интерфейсы

Рассмотрим тип объекта для описания объектов, имеющих метод `id`:

```
type Identifiable = {
    id(): string
}
```

Имея такой тип, мы можем определить функцию, которая ищет элемент по идентификатору:

```
const findById = (elements: Identifiable[], id: string) => {
  for (const e of elements) if (e.id() === id) return e;
    return undefined;
}
```

Чтобы гарантировать, что класс является подтипом этого типа, мы можем определить класс с ключевым словом `implements`:

```
class Person implements Identifiable {
  #name: string
  #id: string
  constructor(name: string, id: string) { this.#name = name; this.#id = id; }
  id() { return this.#id }
}
```

Теперь TypeScript проверяет, что наш класс действительно предоставляет метод `id` с правильными типами.

> **Примечание.** Ничего больше конструкция `implements` не делает. Если ее опустить, то `Person` все равно останется подтипом `Identifiable` в силу структурной типизации.

Существует альтернативный синтаксис для типов объектов, который покажется более знакомым программистам на Java и C#:

```
interface Identifiable {
    id(): string
}
```

В старых версиях TypeScript типы объектов были более ограничены, чем интерфейсы. Теперь они взаимозаменяемы.

Есть два мелких отличия. Один интерфейс может расширять другой:

```
interface Employable extends Identifiable {
  salary(): number
}
```

В объявлениях типов вместо этого используется пересечение типов:

```
type Employable = Identifiable & {
  salary(): number
}
```

Интерфейсы, в отличие от объектов, можно определять частями. Можно написать

```
interface Employable {
  id(): string
}
```

а где-то в другом месте

```
interface Employable {
  salary(): number
}
```

Эти части объединяются. Для объявления `type` такое объединение не производится. Вопрос о том, полезна ли эта возможность, – предмет споров.

> **Примечание.** В TypeScript интерфейс может расширять класс. Тогда он включает все свойства типа экземпляра этого класса. Например,
>
> `interface Point3D extends Point { z: number }`
>
> имеет поля и методы `Point`, а также свойство `z`.
> Вместо такого интерфейса можно использовать тип пересечения
>
> `type Point3D = Point & { z: number }`

13.11. Индексные свойства

Иногда мы хотим использовать объекты с произвольными свойствами. В TypeScript необходимо применять *индексную сигнатуру*, чтобы сообщить компилятору о том, что произвольные свойства допустимы. Синтаксически это выглядит так:

```
type Dictionary = {
  creator: string,
  [arg: string]: string | string[]
}
```

Имя переменной индексного аргумента (здесь – `arg`) несущественно, но должно присутствовать.

У каждого экземпляра `Dictionary` имеется свойство `creator` и сколько угодно других свойств, значения которых имеют тип строки или массива строк.

```
const dict: Dictionary = { creator: 'Pierre' }
dict.hello = ['bonjour', 'salut', 'allô']
let str = 'world'
dict[str] = 'monde'
```

> **Предостережение.** Типы явно заданных свойств должны быть подтипами индексного типа. Следующий код ошибочен:
>
> ```
> type Dictionary = {
>   created: Date, // ошибка - не string  и не string[]
>   [arg: string]: string | string[]
> }
> ```
>
> Иначе было бы невозможно проверить допустимость присваивания `dict[str]` при произвольном значении `str`.

Можно также описать массивоподобные типы с целочисленными значениями индексов:

```
type ShoppingList = {
  created: Date,
  [arg: number] : string
}

const list: ShoppingList = {
  created: new Date()
}
list[0] = 'eggs'
list[1] = 'ham'
```

13.12. Более сложные параметры функций 

В следующих разделах мы покажем, как записывать аннотации для факультативных, подразумеваемых по умолчанию, прочих и деструктурированных параметров. А затем обратимся к «перегрузке» – заданию нескольких вариантов совокупности типов параметров и возвращаемого значения для одной функции.

13.12.1. Факультативные, подразумеваемые по умолчанию и прочие параметры

Рассмотрим JavaScript-функцию

```
const average = (x, y) => (x + y) / 2 // JavaScript
```

В JavaScript нужно помнить о том, что кто-то может вызвать `average(3)`, и тогда функция вычислит `(3 + undefined) / 2`, или `NaN`. В TypeScript это не проблема. Нельзя вызвать функцию, не указав все обязательные аргументы.

Однако программисты на JavaScript часто передают факультативные параметры. В нашей функции `average` второй параметр может быть факультативным:

```
const average = (x, y) => y === undefined ? x : (x + y) / 2 // JavaScript
```

В TypeScript факультативные параметры помечаются знаком `?`, например:

```
const average = (x: number, y?: number) => y === undefined ? x : (x + y) / 2
  // TypeScript
```

Факультативные параметры должны располагаться после обязательных.

Как и в JavaScript, в TypeScript возможны параметры по умолчанию:

```
const average = (x = 0, y = x) => (x + y) / 2 // TypeScript
```

Здесь типы параметров выводятся из типов значений по умолчанию.

«Прочие» параметры работают точно так же, как в JavaScript. Они аннотируются как массив:

```
const average = (first = 0, ...following: number[]) => {
  let sum = first
  for (const value of following) { sum += value }
    return sum / (1 + following.length)
}
```

Эта функция имеет тип

```
(arg1: number, ...arg2: number[]) => number
```

13.12.2. Деструктуризация параметров

В главе 3 мы видели функции, которые вызываются с «объектом конфигурации»:

```
const result = mkString(elements,
  { separator: ', ', leftDelimiter: '(', rightDelimiter: ')' })
```

При реализации такой функции мы, конечно, можем завести параметр для объекта конфигурации:

```
const mkString = (values, config) =>
  config.leftDelimiter + values.join(config.separator) + config.rightDelimiter
```

Или применить деструктуризацию для объявления трех параметров-переменных:

```
const mkString = (values, { separator, leftDelimiter, rightDelimiter }) =>
leftDelimiter + values.join(separator) + rightDelimiter
```

В TypeScript нужно добавить типы. Однако очевидный подход не работает:

```
const mkString = (values: unknown[], { // TypeScript
  separator: string,
  leftDelimiter: string, // ошибка - повторяющийся идентификатор
  rightDelimiter: string // ошибка - повторяющийся идентификатор
}) => leftDelimiter + values.join(separator) + rightDelimiter
```

Синтаксис аннотаций типов в TypeScript вступает в конфликт с синтаксисом деструктуризации. В JavaScript (а следовательно, и в TypeScript) можно добавлять имена переменных после имен свойств:

```
const mkString = (values, { // JavaScript
  separator: sep,
  leftDelimiter: left,
  rightDelimiter: right
}) => left + values.join(sep) + right
```

Чтобы корректно специфицировать типы, добавьте аннотацию типа ко всему объекту конфигурации:

```
const mkString = (values: unknown[], // TypeScript
  { separator, leftDelimiter, rightDelimiter }
    : { separator: string, leftDelimiter: string, rightDelimiter: string })
  => leftDelimiter + values.join(separator) + rightDelimiter
```

В главе 3 мы также видели аргументы по умолчанию для каждого конфигурационного параметра. Вот пример функции с аргументами по умолчанию:

```
const mkString = (values: unknown[], // TypeScript
  { separator = ',', leftDelimiter = '[', rightDelimiter = ']' }
    : { separator?: string, leftDelimiter?: string, rightDelimiter?: string })
  => leftDelimiter + values.join(separator) + rightDelimiter
```

Заметим, что при таких умолчаниях тип немного изменился – теперь все свойства факультативны.

13.12.3. Вариантность типа функции

В разделе 13.7.3 мы видели, что массивы ковариантны. Замена типа элементов подтипом приводит к подтипу массива. Например, если `Employee` – подтип `Person`, то `Employee[]` – подтип `Person[]`.

Аналогично типы объектов ковариантны относительно типов свойств. Тип `{ partner: Employee }` является подтипом `{ partner: Person }`.

В этом разделе мы рассмотрим связи между типами функций.

Типы функций *контравариантны* относительно типов параметров. Если заменить тип параметра *супертипом*, то получится подтип. Например, тип

```
type PersonConsumer = (arg1: Person) => void
```

является подтипом

```
type EmployeeConsumer = (arg1: Employee) => void
```

Это означает, что в переменной типа `EmployeeConsumer` можно хранить значение типа `PersonConsumer`:

```
const pc: PersonConsumer = (p: Person) => { console.log(`a person named ${p.name}`) }
const ec: EmployeeConsumer = pc
  // ec может содержать значение подтипа
```

Это присваивание допустимо, потому что `pf`, без сомнения, может принимать экземпляры типа `Employee`. Ведь она же готова обрабатывать и более общие экземпляры типа `Person`.

Что касается типа возвращаемого значения, то тут имеет место ковариантность. Например,

```
type EmployeeProducer = (arg1: string) => Employee
```

является подтипом

```
type PersonProducer = (arg1: string) => Person
```

Следующее присваивание корректно:

```
const ep: EmployeeProducer = (name: string) => ({ name, salary: 0 })
const pp: PersonProducer = ep
  // pp может содержать значение подтипа
```

Вызов `pp('Fred')`, безусловно, возвращает экземпляр `Person`.

Опустив тип последнего параметра в типе функции, мы получим подтип. Например,

```
(arg1: number) => number
```

является подтипом

```
(arg1: number, arg2: number) => number
```

Чтобы понять, почему это так, рассмотрим присваивание

```
const g = (x: number) => 2 * x
  // тип (arg1: number) => number
const f: (arg1: number, arg2: number) => number = g
  // f может содержать значение подтипа
```

Вызывать `f` с двумя аргументами безопасно. Второй аргумент попросту игнорируется.

Аналогично, если сделать параметр факультативным, то получится подтип:

```
const g = (x: number, y?: number) => y === undefined ? x : (x + y) / 2
  // тип (arg1: number, arg2?: number) => number
```

```
const f: (arg1: number, arg2: number) => number = g
  // f может содержать значение подтипа
```

И снова вызывать f с двумя аргументами безопасно.

Наконец, если добавить «прочие» параметры, то получится подтип:

```
let g = (x: number, y: number, ...following: number[]) => Math.max(x, y, ...following)
  // тип: (arg1: number, arg2: number, ...rest: number[]) => number
let f: (arg1: number, arg2: number) => number = g
  // f может содержать значение подтипа
```

И опять можно вызывать f с двумя параметрами.

В табл. 13.1 перечислены все рассмотренные выше правила подтипов.

Таблица 13.1. Формирование подтипов

Действие	Супертип Переменная этого типа...	Подтип ...может содержать значение этого типа
Заменить тип элемента массива подтипом	`Person[]`	`Employee[]`
Заменить тип свойства объекта подтипом	`{ partner: Person }`	`{ partner: Employee }`
Добавить свойство объекта	`{ x: number, y: number }`	`{ x: number, y: number, color: string }`
Заменить тип параметра функции *супертипом*	`(arg1: Employee) => void`	`(arg1: Person) => void`
Заменить тип возвращаемого значения функции подтипом	`(arg1: string) => Person`	`(arg1: string) => Employee`
Опустить последний параметр	`(arg1: number, arg2: number) => number`	`(arg1: number) => number`
Сделать последний параметр факультативным	`(arg1: number, arg2: number) => number`	`(arg1: number, arg2?: number) => number`
Добавить «прочие» параметры	`(arg1: number) => number`	`(arg1: number, ...rest: number[]) => number`

13.12.4. Перегрузка

В JavaScript нередко пишут функции, допускающие гибкий вызов. Например, следующая функция подсчитывает количество вхождений буквы в строку:

```
function count(str, c) { return str.length - str.replace(c, '').length }
```

А что, если имеется массив строк? В JavaScript поведение функции легко обобщить на такой случай:

```
function count(str, c) {
  if (Array.isArray(str)) {
    let sum = 0
    for (const s of str) {
      sum += s.length - s.replace(c, '').length
```

```
    }
    return sum
  } else {
    return str.length - str.replace(c, '').length
  }
}
```

В TypeScript для этой функции необходимо определить тип. Это не очень трудно: `str` является либо строкой, либо массивом строк:

```
function count(str: string | string[], c: string) { . . . }
```

Это работает, потому что в обоих случаях возвращается значение типа `number`. Таким образом, функция имеет тип:

```
(str: string | string[], c: string) => number
```

Но что, если тип возвращаемого значения зависит от типов аргументов? Допустим, что мы не подсчитываем символы, а удаляем их:

```
function remove(str, c) { // JavaScript
  if (Array.isArray(str))
    return str.map(s => s.replace(c, ''))
  else
    return str.replace(c, '')
}
```

Теперь возвращается либо `string`, либо `string[]`.

Но использовать тип объединения `string | string[]` в качестве возвращаемого – не лучшая идея. В выражении

```
const result = remove(['Fred', 'Barney'], 'e')
```

мы хотели бы, чтобы значение `result` имело тип `string[]`, а не тип объединения.

Для этого можно воспользоваться *перегрузкой* функции. JavaScript не разрешает перегружать функции в традиционном смысле этого слова, т. е. писать функции с одинаковым именем, но различными типами параметров. Вместо этого мы перечисляем объявления, которые хотели бы реализовать по отдельности, а затем предоставляем одну реализацию:

```
function remove(str: string, c: string): string
function remove(str: string[], c: string): string[]
function remove(str: string | string[], c: string) {
  if (Array.isArray(str))
    return str.map(s => s.replace(c, ''))
  else
    return str.replace(c, '')
}
```

В случае стрелочных функций синтаксис немного отличается. Аннотировать следует тип переменной, в которой будет храниться функция:

```
const remove: {
  (arg1: string, arg2: string): string
  (arg1: string[], arg2: string): string[]
} = (str: any, c: string) => {
  if (Array.isArray(str))
    return str.map(s => s.replace(c, ''))
  else
    return str.replace(c, '')
}
```

> **Предостережение.** Быть может, в силу исторических причин в синтаксисе такой перегруженной аннотации не используется стрелочный синтаксис для типов функций. Вместо этого синтаксис напоминает объявление `interface`.
> Кроме того, проверка типов для стрелочных функций работает не особенно хорошо. Параметр `str` должен быть объявлен с типом `any`, а не `string | string[]`. К объявлениям со словом `function` TypeScript относится строже и проверяет пути выполнения функции, гарантируя, что для аргументов типа `string` возвращаются результаты типа `string`, а для аргументов типа `string[]` – результаты типа `string[]`.

Для перегруженных методов используется почти такой же синтаксис, как для функций:

```
class Remover {
  c: string
  constructor(c: string) { this.c = c }

  removeFrom(str: string): string
  removeFrom(str: string[]): string[]
  removeFrom(str: string | string[]) {
    if (Array.isArray(str))
      return str.map(s => s.replace(this.c, ''))
    else
      return str.replace(this.c, '')
  }
}
```

13.13. Обобщенное программирование

Объявление класса, типа или функции называется *обобщенным*, если вместо типов в нем используются *параметрические типы*, которые не заданы, а могут быть подставлены позже. Например, в TypeScript стандартный тип `Set<T>` имеет параметрический тип `T`, что позволяет создавать множества элементов любого типа, например `Set<string>` или `Set<Point>`. В следующих разделах показано, как устроена работа с обобщенными типами в TypeScript.

13.13.1. Обобщенные классы и типы

Ниже приведен пример обобщенного класса. В его экземплярах хранятся пары ключ-значение:

```
class Entry<K, V> {
  key: K
  value: V
  constructor(key: K, second: V) {
    this.key = key
    this.value = value
  }
}
```

Как видим, параметрические типы K и V указываются в угловых скобках после имени класса. В определениях полей и конструктора параметрические типы используются как типы.

Мы *конкретизируем* обобщенный класс, подставляя конкретные типы вместо параметрических. Например, Entry<string, number> – обыкновенный класс с полями типа string и number.

Обобщенным называется тип с одним или несколькими параметрическими типами, например:

```
type Pair<T> = { first: T, second: T }
```

> **Примечание.** Для параметрического типа можно задать умолчание, например:
>
> type Pair<T = any> = { first: T, second: T }
>
> Тогда тип Pair – то же самое, что Pair<any>.

TypeScript предоставляет обобщенные формы классов Set, Map и WeakMap, которые были описаны в главе 7. Нужно только указать типы элементов:

```
const salaries = new Map<Person, number>()
```

Типы также можно вывести из аргументов конструктора. Например, для следующего отображения выводится тип Map<string, number>:

```
const weekdays = new Map(
  [['Mon', 0], ['Tue', 1], ['Wed', 2], ['Thu', 3], ['Fri', 4], ['Sat', 5], ['Sun', 6]])
```

> **Примечание.** Обобщенный класс Array<T> – в точности то же самое, что тип T[].

13.13.2. Обобщенные функции

Как обобщенный класс – это класс с параметрическими типами, так и *обобщенная функция* – это функция с параметрическими типами. Ниже приведен пример функции с одним параметрическим типом. Эта функция подсчитывает, сколько раз значение target встречается в массиве arr.

```
function count<T>(arr: T[], target: T) {
  let count = 0
  for (let e of arr) if (e === target) count++
  return count
}
```

Использование параметрического типа гарантирует, что тип массива такой же, как тип искомого значения.

```
let digits = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
let result = count(digits, 5) // OK
result = count(digits, 'Fred') // ошибка типизации
```

Параметрические типы обобщенной функции всегда располагаются перед открывающей скобкой, которая начинает список параметров функции. Обобщенная стрелочная функция выглядит следующим образом:

```
const count = <T>(arr: T[], target: T) => {
  let count = 0
  for (let e of arr) if (e === target) count++
  return count
}
```

Эта функция имеет тип

```
<T> (arg1: T[], arg2: T) => number
```

При вызове обобщенной функции необязательно задавать параметрические типы. Они выводятся автоматически из типов аргументов. Например, в вызове `count(digits, 5)` типом `digits` будет `number[]`, и TypeScript может вывести, что `T` – это `number`.

При желании можно задать тип и явно, поместив его перед аргументами, например:

```
count<number>(digits, 4)
```

Изредка это приходится делать, если TypeScript выводит не те типы, которые вам нужны. Пример будет приведен в следующем разделе.

13.13.3. Ограничения на типы

Иногда параметрические типы обобщенного класса должны удовлетворять некоторым требованиям. Они выражаются с помощью *ограничений на типы* (type bound).

Рассмотрим следующую функцию, которая возвращает хвост (все элементы, кроме первого) своего аргумента:

```
const tail = <T>(values: T) => values.slice(1) // ошибка
```

Этот подход не годится, потому что TypeScript не знает, имеет ли `values` метод `slice`. Зададим ограничение на тип:

```
const tail = <T extends { slice(from: number, to?: number): T }>(values: T) =>
  values.slice(1) // OK
```

Это ограничение на тип гарантирует, что вызов `values.slice(1)` допустим. Заметим, что ключевое слово `extends` в ограничении на тип в действительности означает «подтип» – проектировщики TypeScript просто воспользовались уже существующим ключевым словом, вместо того чтобы вводить новое слово или символ.

Теперь можно вызвать

```
let result = tail([1, 7, 2, 9]) // присваивает result значение [7, 2, 9]
```

или

```
let greeting = 'Hello'
console.log(tail(greeting)) // отображается ello
```

Разумеется, типу, используемому как ограничение, можно присвоить имя:

```
type Sliceable<T> = { slice(from: number, to?: number): T }
const tail = <T extends Sliceable<T>>(values: T) => values.slice(1)
```

Например, тип `number[]` является подтипом `Sliceable<number[]>`, поскольку метод `slice` возвращает экземпляр типа `number[]`. Аналогично `string` – подтип `Sliceable<string>`.

> **Предостережение.** Попытка вызвать
>
> `console.log(tail('Hello')) // ошибка`
>
> заканчивается ошибкой компиляции – тип `'Hello'` не является подтипом `Sliceable<'Hello'>`. Проблема в том, что `'Hello'` одновременно является экземпляром литерального типа `'Hello'` и типа `string`. TypeScript выбирает литеральный тип, потому что он более специфический, и проверка типа не проходит. Чтобы решить эту проблему, нужно явно конкретизировать обобщенную функцию:
>
> `console.log(tail<string>('Hello')) // OK`
>
> или использовать утверждение типа:
>
> `console.log(tail('Hello' as string))`

13.13.4. Стирание

При трансляции TypeScript на обычный JavaScript все типы стираются. Поэтому вызов

```
let newlyCreated = new T()
```

некорректен. На этапе выполнения нет никакого типа `T`.

Для конструирования объектов произвольных типов необходимо использовать функцию-конструктор, например:

```
const fill = <T>(ctor: { new() : T }, n: number) => {
  let result: T[] = []
  for (let i = 0; i < n; i++)
    result.push(new ctor())
  return result
}
```

Заметим, что тип `ctor` – функция, которую можно вызывать с ключевым словом `new` и которая возвращает значение типа `T`. Это конструктор. Данный конкретный конструктор не имеет аргументов.

При вызове функции fill задается имя класса:

```
const dates = fill(Date, 10)
```

Выражение Date – функция-конструктор. В JavaScript класс – это просто «синтаксический сахар» для функции-конструктора с прототипом.

Аналогично нельзя выполнять обобщенную проверку с помощью instanceof. Следующий код не работает:

```
const filter = <T>(values: unknown[]) => {
  let result: T[] = []
  for (const v of values)
    if (v instanceof T) // Error
      result.push(v)
  return result
}
```

Решение опять же состоит в том, чтобы передать конструктор:

```
const filter = <T>(values: unknown[], ctor: new (...args: any[]) => T) => {
  let result: T[] = []
  for (const v of values)
    if (v instanceof ctor) // OK - правая часть instanceof является конструктором
      result.push(v)
    return result
}
```

Вот пример вызова:

```
const pointsOnly = filter([3, 4, new Point(3, 4), Point.origin], Point)
```

Отметим, что в этом случае конструктор принимает произвольные аргументы.

> **Предостережение.** Проверка с помощью instanceof работает только для *класса*. Нет никакой возможности во время выполнения проверить, является ли значение экземпляром типа или интерфейса.

13.13.5. Вариантность обобщенных типов

Рассмотрим обобщенный тип

```
type Pair<T> = { first: T, second: T }
```

Допустим, что у нас имеется тип Person и его подтип Employee. Какой должна быть связь между Pair<Person> и Pair<Employee>?

Теория типов предлагает три возможности: ковариантные типы (обобщенный тип изменяется в том же направлении, что параметр), контравариантные (изменяется в противоположном направлении) и инвариантные (между обобщенными типами нет никакой связи).

В Java переменные-типы всегда инвариантны, но связь можно выразить с помощью метасимволов, например `Pair<? extends Person>`. В C# можно выбрать вариантность параметрических типов: `Entry<out K, in V>`. В TypeScript нет похожих механизмов.

Вместо этого, принимая решение о том, является ли конкретизация обобщенного типа подтипом другого типа, TypeScript просто подставляет фактические типы и сравнивает получившиеся необобщенные типы.

Например, при сравнении `Pair<Person>` и `Pair<Employee>` подстановка типов `Person` и `Employee` дает

```
{ first: Person, second: Person }
```

и

```
{ first: Employee, second: Employee }
```

В результате тип `Pair<T>` ковариантен относительно `T`. Это некорректно (см. упражнение 15). Однако, как было сказано в разделе 13.7.3, эта некорректность – сознательное проектное решение.

Рассмотрим еще один пример, который иллюстрирует контравариантность:

```
type Formatter<T> = (arg1: T) => string
```

Для сравнения `Formatter<Person>` и `Formatter<Employee>` подставим типы, а затем сравним результаты

```
(arg1: Person) => string
```

и

```
(arg1: Employee) => string
```

Поскольку параметрические типы-функции контравариантны, таков же и параметрический тип `T` в `Formatter<T>`. Это поведение корректно.

13.13.6. Условные типы

Условный тип имеет вид *T* `extends` *U* `?` *V* `:` *W*, где *T*, *U*, *V* и *W* – типы или параметрические типы. Например:

```
type ExtractArray<T> = T extends any[] ? T : never
```

Если `T` – массив, то `ExtractArray<T>` совпадает с `T`. В противном случае он совпадает с `never` – типом, не имеющим ни одного экземпляра.

Сам по себе этот тип не очень полезен. Но его можно использовать для выделения типов из объединений:

```
type Data = string | string[] | number | number[]
type ArrayData = ExtractArray<Data> // тип string[] | number[]
```

Для альтернатив `string` и `number` `ExtractArray` дает тип `never`, который просто отбрасывается.

Теперь предположим, что имеется просто тип элемента. Такая конструкция не работает:

```
type ArrayOf<T> = T extends U[] ? U : never // ошибка - U не определено
```

Вместо этого нужно использовать ключевое слово `infer`:

```
type ArrayOf<T> = T extends (infer U)[] ? U : never
```

Здесь мы проверяем, верно ли, что `T` расширяет `X[]` для некоторого `X`, и если да, то привязываем `U` к `X`. В случае применения к типу объединения все немассивы отбрасываются, а каждый массив заменяется типом своего элемента. Например, `ArrayOf<Data>` равно `number | string`.

13.13.7. Отображаемые типы

Еще один способ задания индексов дают *отображаемые типы*. Если имеется тип объединения строки, целого числа и символьных литералов, то можно определить индексы следующим образом:

```
type Point = {
  [propname in 'x'|'y']: number
}
```

Этот тип `Point` имеет два свойства `x` и `y`, оба типа `number`.

> **Предостережение.** Эта нотация похожа на синтаксис индексных свойств (см. раздел 13.11). Однако у отображаемого типа есть только одно отображение и не может быть дополнительных свойств.

Данный пример не очень полезен. Отображаемые типы предназначены для преобразования существующих типов. Имея тип `Employee`, мы можем сделать все его свойства доступными только для чтения:

```
type ReadonlyEmployee = {
  readonly [propname in keyof Employee]: Employee[propname]
}
```

Здесь мы видим два новых синтаксических элемента.

- Тип `keyof` *T* – это тип объединения всех имен свойств типа *T*, т. е. `'name' | 'salary' | ...` в данном примере.
- Тип *T*`[p]` – это тип свойства с именем *p*. Например, `Employee['name']` – тип `string`.

По-настоящему отображаемые типы блистают в сочетании с обобщенными типами. В библиотеке TypeScript определен следующий служебный тип:

```
type Readonly<T> = {
  readonly [propname in keyof T]: T[propname]
}
```

Он помечает все свойства T как readonly.

> **Совет.** Используя Readonly совместно с типом параметра, мы можем сообщить вызывающей стороне, что параметр неизменяемый.
>
> ```
> const distanceFromOrigin = (p: Readonly<Point>) =>
>   Math.sqrt(Math.pow(p.x, 2) + Math.pow(p.y, 2))
> ```

Еще один пример дает служебный тип Pick, который позволяет выбрать подмножество свойств:

```
let str: Pick<string, 'length' | 'substring'> = 'Fred'
  // К str можно применять только свойства length и substring
```

Этот тип определен следующим образом:

```
type Pick<T, K extends keyof T> = {
  [propname in K]: T[propname]
};
```

Заметим, что extends означает «подтип». Тип keyof string – объединение имен всех свойств типа string. Подтипом тогда является это подмножество этих имен.

Можно также удалить модификатор:

```
type Writable<T> = {
    -readonly [propname in keyof T]: T[propname]
}
```

Чтобы добавить или удалить модификатор ?, воспользуйтесь конструкцией ? или -?:

```
type AllRequired<T> = {
  [propname in keyof T]-?: T[propname]
}
```

Упражнения

1. Что описывают следующие типы?

   ```
   (number | string)[]
   number[] | string[]
   [[number, string]]
   [number, string, ...:number[]]
   [number, string, ...:(number | string)[]]
   [number, ...: string[]] | [string, ...: number[]]
   ```

2. Исследуйте различия между функциями, возвращающими типы void и undefined. Могут ли в функции, возвращающей void, встречаться предложения return? А как насчет возврата undefined или null? Должна ли функция, возвращающая тип undefined, иметь предложение return или может вернуть undefined неявно?

3. Перечислите типы всех функций класса `Math`.
4. В чем разница между типами `object`, `Object` и `{}`?
5. Опишите различия между типами

```
type MaybeColoredPoint = {
 x: number,
 y: number,
 color?: string
}
```

и

```
type PerhapsColoredPoint = {
 x: number,
 y: number,
 color: string | undefined
}
```

6. Рассмотрим тип

```
type Weekday = 'Mon' | 'Tue' | 'Wed' | 'Thu' | 'Fri' | 'Sat' | 'Sun'
```

Является ли `Weekday` подтипом `string` или наоборот?

7. Какая связь тип–подтип имеет место между `number[]` и `unknown[]`? А между `{ x: number, y: number }` и `{ x: number | undefined, y: number | undefined }`? А между `{ x: number, y: number }` и `{ x: number, y: number, z: number }`?
8. Какая связь тип–подтип имеет место между `(arg: number) => void` и `(arg: number | undefined) => void`? А между `() => unknown` и `() => number`? А между `() => number` и `(number) => void`?
9. Какая связь тип–подтип имеет место между `(arg1: number) => number` и `(arg1: number, arg2?: number) => number`?
10. Напишите функцию

```
const act = (x: { bark(): void } | { meow(): void }) => . . .
```

которая вызывает `bark` или `meow` от имени `x`. Чтобы различить варианты, воспользуйтесь оператором `in`.

11. Покажите, что ковариантность объектов некорректна. Используйте типы

```
type Colored = { color: string }
type MaybeColored = { color: string | undefined }
```

Как и в случае массивов (см. раздел 13.7.3), определите две переменные, по одной каждого типа, ссылающиеся на одно и то же значение. Создайте ситуацию, демонстрирующую «дыру» в системе проверки типов, для чего модифицируйте свойство `color` одной переменной и прочитайте это свойство с помощью другой переменной.

12. В разделе 13.11 «Индексные свойства» мы видели, что невозможно написать объявление

```
type Dictionary = {
 created: Date, // ошибка - не string и не string[]
```

```
 [arg: string]: string | string[]
}
```

Сможете ли вы обойти эту проблему с помощью типа пересечения?

13. Рассмотрим следующий тип из раздела 13.11:

```
type ShoppingList = {
 created: Date,
 [arg: number] : string
}
```

Почему следующий код не компилируется?

```
const list: ShoppingList = {
 created: new Date()
}
list[0] = 'eggs'
const more = ['ham', 'hash browns']
for (let i in arr)
 list[i + 1] = arr[i]
```

А почему компилируется этот код?

```
for (let i in arr)
 list[i] = arr[i]
```

14. Приведите примеры пар супертип–подтип для каждой строки в табл. 13.1, отличающиеся от представленных в таблице. Для каждой пары продемонстрируйте, что в переменной супертипа может храниться экземпляр подтипа.
15. Обобщенный класс `Pair<T>`из раздела 13.13.5 ковариантен относительно `T`. Покажите, что это некорректно. Как и в случае массивов (см. раздел 13.7.3), определите две переменные, типа `Pair<Person>` и `Pair<Employee>`, ссылающиеся на одно и то же значение. Измените значение с помощью одной переменной, так чтобы при чтении его из другой переменной возникла ошибка на этапе выполнения.
16. Дополните недостающий код обобщенной функции

```
const last = <. . .> (values: T) => values[values.length - 1]
```

так чтобы следующие вызовы были допустимы:

```
const str = 'Hello'
console.log(last(str))
console.log(last([1, 2, 3]))
console.log(last(new Int32Array(1024)))
```

Указание. Потребуйте, чтобы тип `T` обладал свойством `length` и индексным свойством. Каким должен быть возвращаемый тип индексного свойства?

Предметный указатель

A

B

C

D

E

F