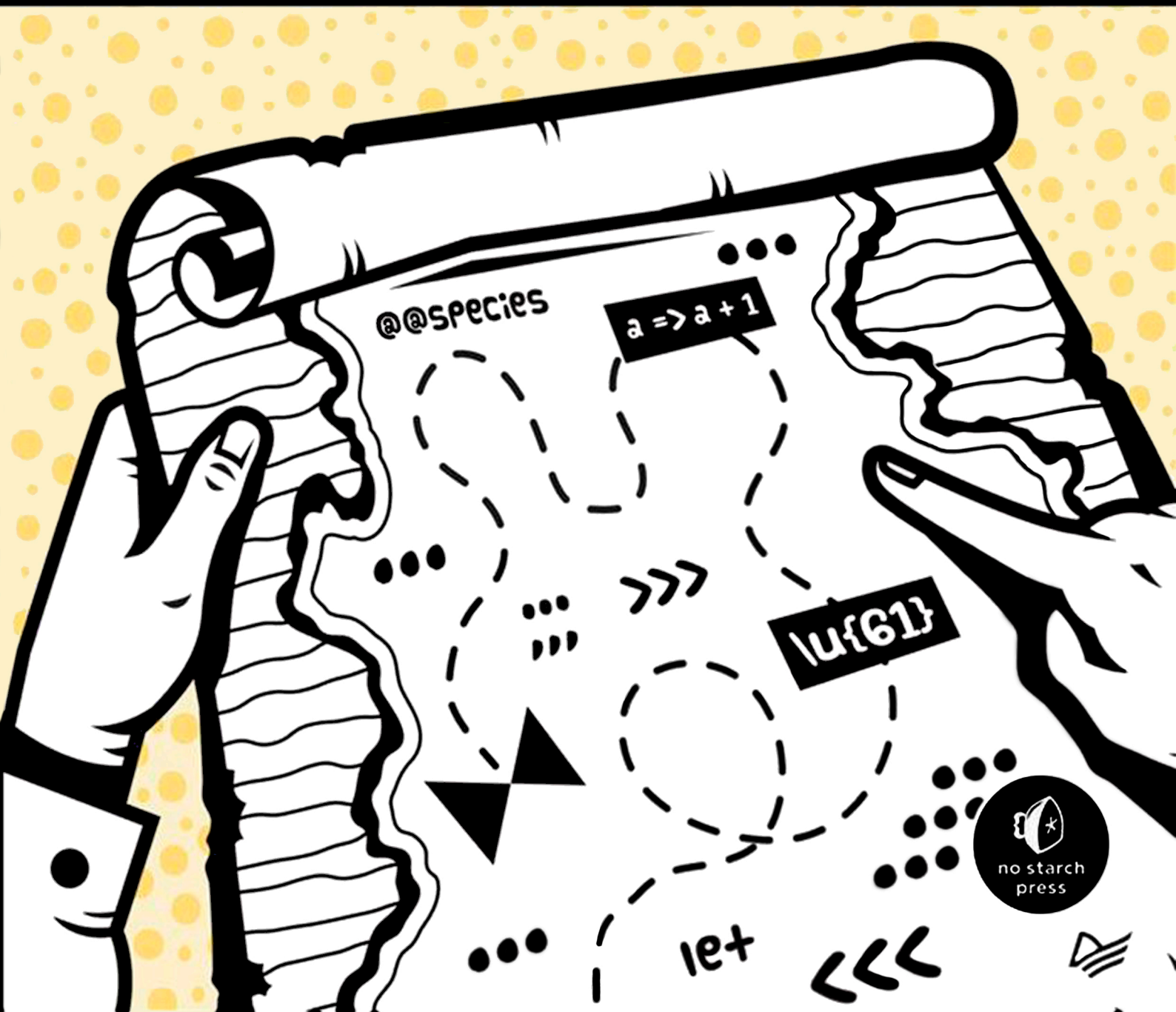


ECMAScript 6

для разработчиков

НИКОЛАС ЗАКАС



UNDERSTANDING ECMAScript 6

The Definitive Guide for JavaScript Developers

by Nicholas C. Zakas

НИКОЛАС ЗАКАС

ECMAScript 6 для разработчиков

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск
2017

ББК 32.988.02-018

УДК 004.738.5

3-18

Закас Н.

3-18 ECMAScript 6 для разработчиков. — СПб.: Питер, 2017. — 352 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-03037-3

Познакомьтесь с радикальными изменениями в языке JavaScript, которые произошли благодаря новому стандарту ECMAScript 6. Николас Закас — автор бестселлеров и эксперт-разработчик — создал самое полное руководство по новым типам объектов, синтаксису и интересным функциям. Каждая глава содержит примеры программ, которые будут работать в любой среде JavaScript и познакомят вас с новыми возможностями языка. Прочитав эту книгу, вы узнаете о том, чем полезны итераторы и генераторы, чем ссылочные функции отличаются от обычных, какие дополнительные опции позволяют работать с данными, о наследовании типов, об асинхронном программировании, о том, как модули меняют способ организации кода, и многом другом.

Более того, Николас Закас заглядывает в будущее, рассказывая про изменения, которые появятся в ECMAScript 7. Неважно, являетесь вы веб-разработчиком или работаете с node.js, в этой книге вы найдете самую необходимую информацию, позволяющую эффективно использовать все возможности ECMAScript 6.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018

УДК 004.738.5

Права на издание получены по соглашению с No Starch Press. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1593277574 англ.

ISBN 978-5-496-03037-3

© 2016 by Nicholas C. Zakas

© Перевод на русский язык ООО Издательство «Питер», 2017

© Издание на русском языке, оформление ООО Издательство «Питер», 2017

© Серия «Библиотека программиста», 2017

Предисловие	14
Благодарности	16
Введение	17
История ECMAScript 6	17
О книге	18
Совместимость с браузерами и Node.js	18
Кому адресована книга	18
Обзор содержания	18
Используемые соглашения	19
От издательства	20
Глава 1. Блочные привязки	21
Объявление и местоположение переменных	21
Объявления на уровне блока	22
Объявления <code>let</code>	22
Повторное объявление недопустимо	23
Объявления <code>const</code>	24
Временная мертвая зона	25
Блочные привязки в циклах	26
Функции в циклах	26
Объявления <code>let</code> в циклах	27
Объявления <code>const</code> в циклах	28
Блочные привязки на глобальном уровне	29
Новые приемы, появившиеся с введением блочных привязок	30
В заключение	30
Глава 2. Строки и регулярные выражения	32
Улучшенная поддержка Юникода	32
Кодовые пункты UTF-16	32
Метод <code>codePointAt()</code>	33
Метод <code>String.fromCodePoint()</code>	34
Метод <code>normalize()</code>	34
Флаг <code>u</code> в регулярных выражениях	36
Другие изменения в поддержке строк	37
Методы идентификации подстрок	37

Содержание	6
Метод <code>repeat()</code>	38
Другие изменения в регулярных выражениях	39
Флаг <code>y</code> в регулярных выражениях	39
Создание копий регулярных выражений	41
Свойство <code>flags</code>	42
Литералы шаблонов	43
Основной синтаксис	43
Многострочный текст	44
Подстановка значений	45
Теги шаблонов	46
В заключение	49
Глава 3. Функции	50
Функции со значениями параметров по умолчанию	50
Имитация значений параметров по умолчанию в ECMAScript 5	50
Значения параметров по умолчанию в ECMAScript 6	51
Как значения параметров по умолчанию влияют на объект <code>arguments</code>	52
Выражения в параметрах по умолчанию	54
Временная мертвая зона для параметров по умолчанию	55
Неименованные параметры	57
Неименованные параметры в ECMAScript 5	57
Остаточные параметры	58
Дополнительные возможности конструктора <code>Function</code>	59
Оператор расширения	60
Свойство <code>name</code>	61
Выбор соответствующих имен	61
Специальные случаи свойства <code>name</code>	62
Двойственная природа функций	62
Как в ECMAScript 5 определить, каким способом вызвана функция	63
Метасвойство <code>new.target</code>	64
Функции уровня блоков	65
Когда использовать функции уровня блока	66
Функции уровня блока в нестрогом режиме	66
Стрелочные функции	67
Синтаксис стрелочных функций	68
Создание выражений немедленно вызываемых функций	69
Отсутствие привязки <code>this</code>	70
Стрелочные функции и массивы	72
Отсутствие привязки <code>arguments</code>	72
Идентификация стрелочных функций	72
Оптимизация хвостовых вызовов	73
Отличия хвостовых вызовов в ECMAScript 6	73
Как использовать оптимизацию хвостовых вызовов	75
В заключение	76
Глава 4. Расширенные возможности объектов	77
Категории объектов	77
Расширение синтаксиса литералов объектов	77
Сокращенный синтаксис инициализации свойств	78
Сокращенный синтаксис определения методов	78
Вычисляемые имена свойств	79
Новые методы	80

Метод <code>Object.is()</code>	80
Метод <code>Object.assign()</code>	81
Дубликаты свойств в литералах объектов	83
Порядок перечисления собственных свойств	84
Расширения в прототипах	85
Смена прототипа объекта	85
Простой доступ к прототипу с помощью ссылки <code>super</code>	86
Формальное определение метода	88
В заключение	89
Глава 5. Деструктуризация для упрощения доступа к данным	91
Какие выгоды дает деструктуризация?	91
Деструктуризация объектов	92
Присваивание с деструктуризацией	92
Значения по умолчанию	93
Присваивание локальным переменным с другими именами	94
Деструктуризация вложенных объектов	95
Деструктуризация массивов	97
Присваивание с деструктуризацией	97
Значения по умолчанию	98
Деструктуризация вложенных массивов	99
Остаточные элементы	99
Смешанная деструктуризация	100
Деструктурированные параметры	101
Деструктурированные параметры являются обязательными	102
Значения по умолчанию для деструктурированных параметров	103
В заключение	103
Глава 6. Символы и символьные свойства	104
Создание символов	104
Использование символов	105
Совместное использование символов	106
Приведение типов для символов	107
Извлечение символьных свойств	108
Экспортирование внутренних операций в виде стандартных символов	109
Метод <code>Symbol.hasInstance</code>	110
Свойство <code>Symbol.isConcatSpreadable</code>	111
Свойства <code>Symbol.match</code> , <code>Symbol.replace</code> , <code>Symbol.search</code> и <code>Symbol.split</code>	112
Метод <code>Symbol.toPrimitive</code>	114
Свойство <code>Symbol.toStringTag</code>	116
Свойство <code>Symbol.unscopables</code>	118
В заключение	119
Глава 7. Множества и ассоциативные массивы	121
Множества и ассоциативные массивы в ECMAScript 5	121
Недостатки обходных решений	122
Множества в ECMAScript 6	123
Создание множеств и добавление элементов	123
Удаление элементов	125
Преобразование множества в массив	127
Множества со слабыми ссылками	128
Ассоциативные массивы в ECMAScript 6	130
Методы ассоциативных массивов	130

Инициализация ассоциативных массивов	131
Метод <code>forEach()</code> ассоциативных массивов	132
Ассоциативные массивы со слабыми ссылками	133
В заключение	137
Глава 8. Итераторы и генераторы	138
Проблемы использования циклов	138
Что такое итераторы?	139
Что такое генераторы?	140
Выражения функций-генераторов	141
Методы-генераторы объектов	142
Итерируемые объекты и циклы <code>for-of</code>	142
Доступ к итератору по умолчанию	143
Создание итерируемых объектов	144
Встроенные итераторы	145
Итераторы коллекций	145
Итераторы строк	149
Итераторы <code>NodeList</code>	150
Оператор расширения и итерируемые объекты, не являющиеся массивами	150
Дополнительные возможности итераторов	151
Передача аргументов в итераторы	152
Возбуждение ошибок внутри итераторов	153
Инструкции <code>return</code> в генераторах	154
Делегирование генераторов	155
Асинхронное выполнение заданий	157
Простой инструмент выполнения заданий	158
Выполнение заданий с данными	159
Инструмент асинхронного выполнения заданий	159
В заключение	162
Глава 9. Введение в классы JavaScript	163
Структуры в ECMAScript 5, подобные классам	163
Объявление класса	164
Объявление простого класса	164
В чем преимущества синтаксиса определения классов?	165
Классы-выражения	167
Простой класс-выражение	167
Именованные классы-выражения	167
Классы как сущности первого класса	169
Свойства с методами доступа	170
Вычисляемые имена членов	171
Методы-генераторы	172
Статические члены	173
Наследование в производных классах	174
Затенение методов класса	176
Унаследованные статические члены	177
Производные классы из выражений	178
Наследование встроенных объектов	180
Свойство <code>Symbol.species</code>	181
Использование <code>new.target</code> в конструкторах классов	184
В заключение	185

Глава 10.Расширенные возможности массивов	187
Создание массивов	187
Метод <code>Array.of()</code>	187
Метод <code>Array.from()</code>	189
Новые методы всех массивов	191
Методы <code>find()</code> и <code>findIndex()</code>	191
Метод <code>fill()</code>	192
Метод <code>copyWithin()</code>	193
Типизированные массивы	194
Числовые типы данных	194
Буферы массивов	195
Управление буферами массивов с помощью представлений	195
Сходства типизированных и обычных массивов	201
Общие методы	201
Те же самые итераторы	202
Методы <code>of()</code> и <code>from()</code>	202
Различия типизированных и обычных массивов	203
Различия в поведении	203
Отсутствующие методы	204
Дополнительные методы	204
В заключение	205
Глава 11.Объект <code>Promise</code> и асинхронное программирование	206
Основы асинхронного программирования	206
Модель событий	207
Обратные вызовы	207
Основы объектов <code>Promise</code>	209
Жизненный цикл объекта <code>Promise</code>	209
Создание неустановившихся объектов <code>Promise</code>	211
Создание установившихся объектов <code>Promise</code>	213
Ошибки исполнителя	215
Глобальная обработка отклоненных объектов <code>Promise</code>	216
Обработка отказов в <code>Node.js</code>	216
Обработка отказов в браузерах	218
Составление цепочек из объектов <code>Promise</code>	220
Перехват ошибок	220
Возврат значений в цепочке объектов <code>Promise</code>	221
Возврат объектов <code>Promise</code> в цепочке	222
Обработка сразу нескольких объектов <code>Promise</code>	224
Метод <code>Promise.all()</code>	225
Метод <code>Promise.race()</code>	226
Наследование <code>Promise</code>	227
Выполнение асинхронных заданий с помощью <code>Promise</code>	228
В заключение	231
Глава 12.Прокси-объекты и <code>Reflection API</code>	233
Проблема с массивами	233
Введение в прокси-объекты и <code>Reflection API</code>	234
Создание простого прокси-объекта	235
Проверка свойств с помощью ловушки <code>set</code>	235
Проверка формы объектов с помощью ловушки <code>get</code>	237
Соккрытие свойств с помощью ловушки <code>has</code>	238

Предотвращение удаления свойств с помощью ловушки <code>deleteProperty</code>	239
Ловушки операций с прототипом	241
Как действуют ловушки операций с прототипом	241
Почему поддерживается два набора методов?	242
Ловушки, связанные с расширяемостью объектов	244
Два простых примера	244
Дубликаты методов управления расширяемостью	245
Ловушки операций с дескрипторами свойств	246
Блокирование вызова <code>Object.defineProperty()</code>	246
Ограничения объекта дескриптора	247
Дубликаты методов для операций с дескрипторами	248
Ловушка <code>ownKeys</code>	249
Обработка вызовов функций с помощью ловушек <code>apply</code> и <code>construct</code>	250
Проверка параметров функции	251
Вызов конструкторов без ключевого слова <code>new</code>	253
Переопределение конструкторов абстрактных базовых классов	254
Вызываемые конструкторы классов	255
Отключение прокси-объектов	256
Решение проблемы с массивами	256
Определение индексов массива	257
Увеличение значения <code>length</code> при добавлении новых элементов	258
Удаление элементов при уменьшении значения <code>length</code>	259
Реализация класса <code>MyArray</code>	260
Использование прокси-объекта в качестве прототипа	262
Использование ловушки <code>get</code> в прототипе	263
Использование ловушки <code>set</code> в прототипе	264
Использование ловушки <code>has</code> в прототипе	265
Прокси-объекты как прототипы классов	265
В заключение	268
Глава 13. Инкапсуляция кода в модули	270
Что такое модули?	270
Основы экспортирования	270
Основы импортирования	271
Импортирование единственной привязки	272
Импортирование нескольких привязок	272
Импортирование всего модуля	273
Тонкая особенность импортированных привязок	274
Переименование экспортируемых и импортируемых привязок	274
Значения по умолчанию в модулях	275
Экспортирование значений по умолчанию	275
Импортирование значений по умолчанию	276
Реэкспорт привязки	276
Импортирование без привязок	277
Загрузка модулей	278
Использование модулей в веб-браузерах	278
Разрешение спецификаторов модулей в браузерах	282
В заключение	282

Приложение А. Мелкие изменения в ECMAScript 6	284
Новые приемы работы с целыми числами	284
Идентификация целых чисел	284
Безопасные целые числа	285
Новые математические методы	285
Идентификаторы с символами Юникода	286
Формализованное свойство <code>__proto__</code>	287
Приложение Б. Введение в ECMAScript 7 (2016)	289
Оператор возведения в степень	289
Порядок операций	289
Ограничения операндов	290
Метод <code>Array.prototype.includes()</code>	291
Как используется метод <code>Array.prototype.includes()</code>	291
Сравнение значений	291
Изменение области видимости функций в строгом режиме	292

Николас Закас (Nicholas Zakas) занимается разработкой веб-приложений, в основном клиентской их части, начиная с 2000 года и широко известен своими книгами и лекциями о передовых приемах разработки пользовательского интерфейса. В течение пяти лет оттачивал свой опыт, работая в Yahoo!, где занимал пост ведущего инженера, отвечающего за главную страницу Yahoo!. Автор нескольких книг, включая *Principles of Object-Oriented JavaScript* (No Starch Press, 2014) и *Professional JavaScript for Web Developers* (Wrox, 2012)¹.

¹ Николас Закас «JavaScript для профессиональных веб-разработчиков». — Пер. с англ. — СПб.: Питер, 2015.

О научном редакторе

Юрий Зайцев (Juriy Zaytsev, известен в Сети под псевдонимом *kangax*) — вебразработчик, проживающий в Нью-Йорке. Исследует и пишет о необычной природе JavaScript начиная с 2007 года. Вносит вклад в развитие нескольких открытых проектов, включая Prototype.js, и других популярных проектов, таких как его собственный Fabric.js. Сооснователь компании printio.ru, занимающейся печатью под заказ. В настоящее время работает в Facebook.

Язык ECMAScript 6 вихрем ворвался в мир. Он появился, когда многие уже перестали ждать его, и распространялся быстрее, чем многие успевали знакомиться с ним. У каждого своя история об этом. А вот моя.

В 2013 году я работал в проекте, целью которого была связать iOS с Веб. Это было еще до моего участия в создании проекта Redux и до вступления в сообщество разработчиков программного обеспечения с открытым исходным кодом на JavaScript. В тот момент я упорно пытался освоить принципы разработки веб-приложений и пребывал в страхе перед новым и неизведанным. Наша команда должна была всего за несколько месяцев создать с нуля веб-версию нашего продукта на JavaScript.

Сначала мне казалось невозможным написать на JavaScript что-то более или менее серьезное. Но новый член команды убедил меня, что JavaScript — далеко не игрушечный язык программирования, и я согласился дать ему шанс. Я отбросил все свои предубеждения, открыл MDN и StackOverflow и впервые приступил к детальному изучению JavaScript. Простота языка, которую я обнаружил, очаровала меня. Один из коллег научил меня пользоваться такими инструментами, как linter и bundler. Однажды, спустя несколько недель, я проснулся и понял, что обожаю писать на JavaScript.

Но, как известно, совершенных языков не существует. Я не видел частых обновлений, к которым привык, работая с другими языками. Единственное существенное обновление в JavaScript за десятилетие — ECMAScript 5 — оказалось простой чисткой, но даже в этом случае потребовалось несколько лет, чтобы все браузеры реализовали полноценную поддержку обновленной версии. В то время грядущая спецификация ECMAScript 6 (ES6) под кодовым названием *Harmony* была далека от завершения, и казалось, что она выйдет в далеком-далеком будущем. Я тогда думал, что, возможно, лишь лет через десять смогу написать первый код на ES6.

На тот момент существовало несколько экспериментальных «трансляторов», таких как Google Traceur, которые преобразовывали код на ES6 в код на ES5. Большинство из них имело серьезные ограничения или было жестко завязано на существующий конвейер сборки JavaScript. Но затем появился новый транслятор под названием *bto5*, и все изменилось. Он был прост в установке, хорошо интегрировался с существующими инструментами и производил читаемый программный код. Он распространился, подобно лесному пожару. Транслятор *bto5*, ныне известный под названием *Babel*, позволил использовать новые возможности ES6 еще до того, как работа над спецификацией была завершена. За несколько месяцев ES6 распространился повсюду.

ES6 разделил сообщество в силу ряда причин. Когда эта книга уходила в печать, еще не все его возможности были реализованы в основных браузерах. Наличие этапа сборки выглядит пугающим для начинающих изучать язык. Некоторые библиотеки сопровождаются документацией и примерами на ES6, и у многих возникает сомнение в возможности использовать эти библиотеки в коде на ES5. Это вносит дополнительную путаницу. Многие не ожидали появления новых возможностей в языке, потому что он почти не изменялся прежде. Другие с нетерпением ждали нововведений и сразу приступили к использованию всех новых возможностей, даже когда в этом не было необходимости.

Как раз когда я только набрался некоторого опыта программирования на JavaScript, я почувствовал, что как будто кто-то пытается вырвать коврик из-под моих ног, и теперь я вынужден приступать к изучению нового языка. В течение нескольких месяцев меня мучила неуверенность. Наконец, в канун Рождества я начал читать рукопись этой книги и не смог оторваться от нее. Следующее, что я помню: три часа ночи, все, кто был на вечеринке, уже спят, а я наконец-то начал понимать ES6!

Николас — невероятно талантливый учитель. Он рассказывает о тонкостях настолько просто и понятно, что новые знания аккуратно укладываются в вашей голове. Помимо этой книги он также известен как автор ESLint — инструмента для анализа кода на JavaScript, количество загрузок которого превысило несколько миллионов.

Николас знает JavaScript в таких подробностях, которые многим и не снились. Не упустите свой шанс впитать толику его знаний. Прочитайте эту книгу, и вы тоже почувствуете уверенность в своем понимании ES6.

Дэн Абрамов (Dan Abramov)

Член основной команды проекта React и создатель Redux

Благодарности

Спасибо Дженифер Гриффит-Дельгадо (Jennifer Griffith-Delgado), Элисон Лоу (Alison Law) и всем сотрудникам издательства No Starch Press за поддержку и помощь в создании этой книги. Их понимание и терпение к замедлению темпа работы, когда я серьезно заболел, я никогда не забуду.

Я благодарен за внимательность Юрию Зайцеву, научному редактору этой книги, и доктору Акселу Раушмайеру (Dr. Axel Rauschmayer) за ответы на вопросы, которые помогли мне прояснить некоторые идеи, обсуждаемые в книге.

Спасибо всем, кто прислал свои исправления к предварительной версии книги, выложенной в GitHub: 404, alexuans, Ахмад Али (Ahmad Ali), Радж Ананд (Raj Anand), Аржинкумар (Arjunkumar), Пахлеви Фикри Ауля (Pahlevi Fikri Auliya), Мохсен Азими (Mohsen Azimi), Питер Баконди (Peter Bakondy), Сарбботтам Бандиопадхаяй (Sarbbottam Bandyopadhyay), blacktail, Филип Борисов (Philip Borisov), Ник Боттомли (Nick Bottomley), Этан Браун (Ethan Brown), Джереми Кани (Jeremy Caney), Джейк Чемпион (Jake Champion), Дэвид Чанг (David Chang), Карло Костантини (Carlo Costantini), Аарон Данди (Aaron Dandy), Нильс Деккер (Niels Dequeker), Александр Джинджик (Aleksandar Djindjic), Джо Имз (Joe Eames), Льюис Эллис (Lewis Ellis), Ронен Эльстер (Ronen Elster), Ямунд Фергюсон (Jamund Ferguson), Стивен Фут (Steven Foote), Росс Гербази (Ross Gerbasi), Шон Хиксон (Shaun Hickson), Даррен Хаски (Darren Huskie), jakub-g, kavun, Наванит Кесаван (Navaneeth Kesavan), Дэн Килп (Dan Kielp), Рой Линг (Roy Ling), Роман Ло (Roman Lo), Lonniebiz, Кевин Лозандье (Kevin Lozandier), Джош Любавэй (Josh Lubaway), Мэллори (Mallory), Якуб Наребски (Jakub Nar bski), Робин Покорны (Robin Pokorny), Кайл Поллок (Kyle Pollock), Франческо Понгилуппи (Francesco Pongiluppi), Николас Понирос (Nikolas Poniros), Абдул-Фаттах Попула (AbdulFattah Popoola), Бен Регенспан (Ben Regenspan), Адам Ричеймер (Adam Richeimer), robertd, Марьян Русняк (Marian Rusnak), Пол Салаетс (Paul Salaets), Shidhin, ShMcK, Кайл Симпсон (Kyle Simpson), Игорь Скухарь (Igor Skuhar), Янг Сю (Yang Su), Эрик Санда (Erik Sundahl), Дмитрий Суворов (Dmitri Suvorov), Кевин Суини (Kevin Sweeney), Прайяг Верма (Prayag Verma), Рик Уолдрон (Rick Waldron), Кейл Уорсли (Kale Worsley), Юрий Зайцев и Евгений Зубарев (Eugene Zubarev).

Спасибо также Кейси Виско (Casey Visco) за поддержку этой книги в Patreon.

Основные особенности языка JavaScript определены в стандарте ECMA-262. Этот стандарт описывает язык с названием ECMAScript. Язык, известный как JavaScript и используемый в браузерах и Node.js, в действительности является надмножеством ECMAScript. Браузеры и Node.js расширяют язык дополнительными объектами и методами, но само ядро JavaScript по-прежнему определяется спецификацией ECMAScript. Продолжение развития ECMA-262 было жизненно важным условием успеха JavaScript в целом, и эта книга охватывает изменения в языке, которые были привнесены последним обновлением спецификации ECMAScript 6.

История ECMAScript 6

В 2007 году язык JavaScript оказался на перепутье. Появление популярной технологии Ajax возвестило о начале новой эры динамических веб-приложений, но к тому времени JavaScript не изменялся с момента выхода третьей редакции ECMA-262, опубликованной в 1999 году.

TC-39 — комитет, ответственный за развитие ECMAScript, — собрал воедино рабочий вариант огромной спецификации ECMAScript 4, предусматривавшей крупные и мелкие изменения в языке. В числе изменений были: новый синтаксис, поддержка модулей, классы, классическое наследование, приватные члены объектов, необязательные аннотации типов и многое другое.

Большой объем изменений, предлагавшихся в ECMAScript 4, вызвал раскол в комитете TC-39 — некоторые его члены полагали, что четвертая редакция включает слишком много изменений. Группа лидеров из Yahoo!, Google и Microsoft разработала альтернативный вариант следующей версии ECMAScript, который первоначально назывался ECMAScript 3.1. Номер версии «3.1» должен был подчеркнуть, что эта версия вносит в существующий стандарт небольшие поступательные изменения.

В ECMAScript 3.1 предлагались очень ограниченные изменения в синтаксисе и основной упор делался на введение атрибутов свойств, встроенной поддержки JSON и дополнительных методов у уже существующих объектов. Первые попытки согласовать ECMAScript 3.1 и ECMAScript 4 полностью провалились, потому что два лагеря имели совершенно разные взгляды на то, как должен развиваться язык.

В 2008 году Брендан Эйх, создатель JavaScript, заявил, что комитет TC-39 должен сосредоточить свои усилия на стандартизации ECMAScript 3.1 и отложить существенные изменения синтаксиса и особенностей языка, предложенные в ECMAScript 4, пока не будет стандартизована следующая версия ECMAScript, и все члены комитета должны стремиться свести воедино все лучшее из ECMAScript 3.1 и 4. С этого момента начала свою историю версия ECMAScript под названием Harmony.

В конечном итоге спецификация ECMAScript 3.1 была стандартизована как пятая редакция ECMA-262, которую также часто называют ECMAScript 5. Версия стандарта ECMAScript 4 так и не была выпущена, чтобы избежать путаницы с неудавшейся попыткой. После этого началась работа

над ECMAScript Harmony, и версия стандарта ECMAScript 6 стала первой, созданной в новом духе «гармонии».

Работы над версией ECMAScript 6 были завершены в 2015 году, и спецификация получила официальное название «ECMAScript 2015». (Но в этой книге она по-прежнему называется ECMAScript 6, потому что это название привычнее для разработчиков.) Она определяет весьма широкий диапазон изменений — от совершенно новых объектов до синтаксических конструкций и новых методов в существующих объектах. Самое замечательное, что все изменения в ECMAScript 6 направлены на решение проблем, с которыми разработчики сталкиваются ежедневно.

О книге

Знание и понимание особенностей ECMAScript 6 совершенно необходимы любым разработчикам на JavaScript. Возможности языка, введенные в ECMAScript 6, образуют фундамент, на котором будут строиться JavaScript-приложения в обозримом будущем. Я надеюсь, что, читая эту книгу, вы сможете освоить новые возможности ECMAScript 6 и будете готовы применить их, когда это потребуется.

Совместимость с браузерами и Node.js

Разработчики многих окружений JavaScript, таких как веб-браузеры и Node.js, активно работают над реализацией ECMAScript 6. Эта книга не пытается описать все несоответствия между реализациями; ее цель — рассказать, какое поведение считается корректным с точки зрения спецификации. Поэтому есть вероятность, что ваше окружение JavaScript действует иначе, чем описано в этой книге.

Кому адресована книга

Эта книга задумывалась как руководство для тех, кто уже знаком с JavaScript и ECMAScript 5. Хотя глубокое знание языка не является обязательным условием для работы с этой книгой, однако оно поможет вам понять различия между ECMAScript 5 и 6. В частности, эта книга адресована разработчикам на JavaScript с опытом программирования сценариев для браузеров или Node.js, желающим узнать о последних нововведениях, появившихся в языке.

Эта книга точно не для начинающих, никогда не писавших на JavaScript. Для чтения этой книги необходимо хорошо знать хотя бы основы языка.

Обзор содержания

Все главы и приложения в этой книге охватывают разные аспекты ECMAScript 6. Многие главы начинаются с обсуждения проблем, на решение которых направлены изменения в ECMAScript 6, чтобы вы получили более широкое представление об этих изменениях. Все главы включают примеры программного кода, демонстрирующие новые идеи и синтаксические конструкции.

- В главе 1 «Блочные привязки» рассказывается об операторах `let` и `const`, замещающих оператор `var` на уровне блоков.
- Глава 2 «Строки и регулярные выражения» охватывает дополнительные строковые функции, а также знакомит с шаблонными строками.
- В главе 3 «Функции» обсуждаются различные изменения, коснувшиеся функций, включая стрелочные функции, параметры со значениями по умолчанию, остаточные параметры и некоторые другие особенности.

- В главе 4 «Расширенные возможности объектов» разъясняются изменения в подходах к созданию, изменению и использованию объектов. В числе рассматриваемых тем изменения в синтаксисе литералов объектов и новые методы рефлексии.
- Глава 5 «Деструктуризация для упрощения доступа к данным» знакомит с синтаксисом деструктуризации объектов и массивов, позволяющим расчленять объекты и массивы, используя компактный синтаксис.
- Глава 6 «Символы и символьные свойства» знакомит с идеей символов — новым способом определения свойств. Символы — это новый простой тип данных, который можно использовать для сокрытия (хотя и не полного) свойств и методов объектов.
- В главе 7 «Множества и ассоциативные массивы» описываются новые типы коллекций: `Set`, `WeakSet`, `Map` и `WeakMap`. Эти типы добавляют в обычные массивы гарантию уникальности значений и средства управления памятью, спроектированные специально для JavaScript.
- В главе 8 «Итераторы и генераторы» обсуждается добавление в язык итераторов и генераторов. Эти инструменты открывают новые мощные способы работы с коллекциями данных, недоступные в предыдущих версиях JavaScript.
- Глава 9 «Введение в классы JavaScript» знакомит с первым формализованным понятием классов в JavaScript. Объектно-ориентированная модель в JavaScript часто вызывает путаницу у программистов, перешедших из других языков. Новый дополнительный синтаксис для работы с классами делает JavaScript более доступным для других и выразительным для энтузиастов.
- В главе 10 «Расширенные возможности массивов» описываются изменения, коснувшиеся обычных массивов, и новые способы их использования в JavaScript.
- Глава 11 «Объект Promise и асинхронное программирование» знакомит с объектами асинхронных вычислений (`Promise`) — новым элементом языка. Объекты асинхронных вычислений явились результатом массовых усилий и быстро завоевали популярность благодаря обширной поддержке в библиотеках. Спецификация ECMAScript 6 формализовала эти объекты и сделала их доступными по умолчанию.
- Глава 12 «Прокси-объекты и Reflection API» знакомит с прикладным интерфейсом рефлексии в JavaScript и новыми прокси-объектами, позволяющими перехватывать любые операции с объектом. Прокси-объекты дают разработчикам беспрецедентный контроль над объектами и, как следствие, неограниченные возможности для определения новых шаблонов взаимодействий.
- В главе 13 «Инкапсуляция кода в модули» описывается официальный формат модулей для JavaScript. Его цель состоит в том, чтобы заменить многочисленные нестандартные форматы определения модулей, появившиеся за годы существования языка.
- Приложение А «Мелкие изменения в ECMAScript 6» охватывает прочие изменения в ECMAScript 6, редко используемые или не связанные с более крупными изменениями, описанными в предыдущих главах.
- В приложении Б «Введение в ECMAScript 7 (2016)» описываются три дополнения к стандарту, включенные в редакцию ECMAScript 7, которая не оказывает такого существенного влияния на JavaScript, как ECMAScript 6.

Используемые соглашения

Ниже перечислены типографские соглашения, принятые в этой книге:

- *Курсив* используется для выделения новых терминов и имен файлов.

- Моноширинным шрифтом выделяются фрагменты программного кода.

Кроме того, длинные фрагменты кода оформляются как блоки, набранные моноширинным шрифтом, например:

```
function doSomething() {  
    // пустая  
}
```

Комментарии в блоках кода, находящиеся правее инструкции `console.log()`, описывают вывод, который должен появиться в браузере или в консоли Node.js, например:

```
console.log("Hi"); // "Hi"
```

Если инструкция в блоке кода вызывает ошибку, она отмечается комментарием справа, как показано ниже:

```
doSomething(); // вызывает ошибку
```

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу `comp@piter.com` (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства `www.piter.com` вы найдете подробную информацию о наших книгах.

Традиционно объявление переменных и работа с ними были одной из самых больших сложностей программирования на JavaScript. В большинстве С-подобных языков переменные (более формально называемые *привязками* (bindings), так как связывают имя со значением в определенной области видимости) создаются в точке их объявления. Но в JavaScript дело обстоит иначе. Выбор области видимости, где фактически будет создана переменная, зависит от того, как она объявляется, и ECMAScript 6 предлагает возможности, позволяющие контролировать этот выбор. В данной главе обсуждается, почему классические объявления `var` могут вызывать путаницу, рассказывается о локальных привязках (на уровне блока), появившихся в ECMAScript 6, а затем демонстрируются некоторые практические приемы их использования.

Объявление и местоположение переменных

Объявления переменных с помощью оператора `var` интерпретируются, как если бы они находились в начале функции (или глобальной области видимости, если объявление находится за пределами функции), независимо от того, где фактически находится объявление; этот эффект носит название *поднятие переменных* (*hoisting*). Чтобы понять его суть, рассмотрим следующее определение функции:

```
function getValue(condition) {  
  
    if (condition) {  
        var value = "blue";  
  
        // прочий код  
  
        return value;  
    } else {  
  
        // переменная value существует здесь и имеет значение undefined  
        return null;  
    }  
  
    // переменная value существует здесь и имеет значение undefined  
}
```

Разработчики, не знакомые с JavaScript, могли бы предположить, что переменная `value` создается, только если условие `condition` имеет значение `true`. В действительности, она будет создана в любом случае. За кулисами интерпретатор JavaScript изменит функцию `getValue`, как показано ниже:

```
function getValue(condition) {  
  
    var value;  
  
    if (condition) {  
        value = "blue";  
  
        // прочий код  
  
        return value;  
    } else {  
  
        return null;  
    }  
}
```

Объявление переменной `value` будет поднято в начало функции, а ее инициализация останется на месте. Это означает, что переменная `value` окажется доступна также в ветке `else`, где будет иметь значение `undefined`, потому что не инициализируется в блоке `else`.

Разработчикам, только начинающим использовать JavaScript, часто требуется некоторое время, чтобы привыкнуть к эффекту подъема объявлений переменных, а недопонимание этого уникального явления может приводить к ошибкам. По этой причине в ECMAScript 6 были введены средства ограничения области видимости переменных, чтобы дать разработчикам более полный контроль над жизненным циклом переменных.

Объявления на уровне блока

Объявления на уровне блока создают привязки (переменные), недоступные за пределами блока. *Область видимости блока (block scopes)*, которую также называют *лексической областью видимости (lexical scopes)*, создается в следующих местах:

- внутри функции;
- внутри блока (ограниченного фигурными скобками `{` и `}`).

Области видимости блока поддерживаются во многих C-подобных языках, и чтобы обеспечить ту же гибкость (и единообразие) в JavaScript, они были введены в ECMAScript 6.

Объявления `let`

Объявление `let` имеет тот же синтаксис, что и объявление `var`. В простейшем случае оператор `var` в объявлении переменной можно заменить на `let`, но это ограничит область видимости переменной текущим блоком кода (существуют также другие тонкие отличия, которые обсуждаются ниже в разделе «Временная мертвая зона»). Объявления `let` не поднимаются к началу блока, поэтому их лучше размещать в блоке первыми, чтобы сделать доступными во всем блоке. Например:

```
function getValue(condition) {
  if (condition) {
    let value = "blue";

    // прочий код

    return value;
  } else {

    // переменная value не существует здесь

    return null;
  }
  // переменная value не существует здесь
}
```

Поведение этой версии функции `getValue` полнее соответствует ожидаемому в других С-подобных языках. Так как переменная `value` объявлена с помощью `let` вместо `var`, объявление не поднимается интерпретатором в начало определения функции, и переменная `value` оказывается недоступной за пределами блока `if`. Если `condition` получит значение `false`, переменная `value` не будет объявлена и инициализирована.

Повторное объявление недопустимо

Если идентификатор уже определен в текущей области видимости, его использование в объявлении `let` вызовет ошибку. Например:

```
var count = 30;

let count = 40; // вызовет ошибку
```

В данном примере переменная `count` объявляется дважды: один раз с помощью `var` и второй — с помощью `let`. Так как объявление `let` не позволяет переопределять идентификаторы, уже присутствующие в текущей области видимости, попытка выполнить его вызовет ошибку. С другой стороны, если объявление `let` создает новую переменную с именем, уже объявленным во внешней области видимости, это не приведет к ошибке, как демонстрирует следующий фрагмент:

```
var count = 30;

if (condition) {

  // не вызовет ошибку
  let count = 40;

  // прочий код
}
```

Это объявление `let` не вызовет ошибку, потому что создает новую переменную с именем `count` внутри инструкции `if`, а не в окружающем ее блоке. Внутри блока `if` эта новая переменная закроет доступ к глобальной переменной `count` до конца блока.

Объявления `const`

Привязки в ECMAScript 6 можно также объявлять с помощью `const`. Такие привязки считаются константами, то есть их значения невозможно изменить после инициализации. По этой причине каждая привязка `const` должна включать значение для инициализации, как показано в следующем примере:

```
// допустимая константа
const maxItems = 30;
```

```
// синтаксическая ошибка: отсутствует значение для инициализации
const name;
```

Привязка `maxItems` инициализируется, поэтому объявление `const` не вызовет ошибки. Но если попытаться выполнить программу, содержащую привязку `name`, она вызовет синтаксическую ошибку, потому что объявление `name` не содержит инициализирующего значения.

Константы и объявления `let`

Область видимости констант, так же как и переменных, объявленных с помощью `let`, ограничивается вмещающим блоком. Это означает, что константа становится недоступной, как только поток выполнения покинет блок, в котором она объявлена, и объявления `const` не поднимаются вверх, как демонстрирует следующий пример:

```
if (condition) {
    const maxItems = 5;

    // прочий код
}

// константа maxItems здесь недоступна
```

В этом фрагменте константа `maxItems` объявляется внутри инструкции `if`. Когда инструкция завершит выполнение, `maxItems` окажется недоступной во внешнем блоке.

Объявление `const`, как и `let`, вызывает ошибку при попытке повторно использовать идентификатор, уже объявленный в текущей области видимости. И не важно, как была объявлена переменная — с помощью `var` (в глобальной области видимости или в области видимости функции) или `let` (в блоке). Например:

```
var message = "Hello!";
let age = 25;

// оба следующих объявления вызовут ошибку
const message = "Goodbye!";
const age = 30;
```

Оба объявления `const` сами по себе допустимы, но из-за присутствия предшествующих им объявлений `var` и `let`, они вызывают синтаксические ошибки.

Несмотря на все перечисленные сходства, `const` и `let` имеют одно существенное отличие. Попытка присвоить новое значение идентификатору, прежде объявленному с помощью `const`, вызовет ошибку в обоих режимах выполнения — строгом и нестрогом:

```
const maxItems = 5;

// вызовет ошибку
maxItems = 6;
```

Подобно константам во многих других языках, существующей переменной `maxItems` нельзя присвоить новое значение. Однако, в отличие от констант в других языках, значение константы можно изменить, если оно является объектом.

Объявление объектов с помощью `const`

Объявление `const` не позволяет изменить привязку, но не значение. Следовательно, объявление объекта с помощью `const` не помешает изменить его. Например:

```
const person = {  
  name: "Nicholas"  
};
```

```
// работает  
person.name = "Greg";
```

```
// вызовет ошибку  
person = {  
  name: "Greg"  
};
```

Здесь создается привязка `person`, инициализированная значением в форме объекта с одним свойством. Это дает возможность изменить `person.name` и не вызвать ошибку, потому что при этом изменяется содержимое объекта, но не сам объект, к которому привязан идентификатор `person`. Попытка присвоить новый объект идентификатору `person` (то есть изменить привязку) вызовет ошибку. Такая тонкость в поведении констант в отношении объектов легко может запутать. Просто помните, что `const` не позволяет изменить привязку, но не само привязанное значение.

Временная мертвая зона

Переменная, объявленная с помощью `let` или `const`, недоступна до ее объявления. Попытка обратиться к ней вызовет ошибку ссылки даже в обычно безопасных операциях, таких как `typeof` в следующей инструкции `if`:

```
if (condition) {  
  console.log(typeof value); // вызовет ошибку  
  let value = "blue";  
}
```

Здесь переменная `value` объявлена и инициализирована с помощью `let`, но эта инструкция никогда не будет выполнена, потому что предыдущая строка вызовет ошибку. Проблема в том, что переменная `value` используется в области, которую в сообществе JavaScript называют *временной мертвой зоной* (*Temporal Dead Zone, TDZ*). В спецификации ECMAScript нет определения термина TDZ, но он часто используется для описания областей, предшествующих привязкам `let` и `const`, где они оказываются недоступными. В этом разделе описываются некоторые тонкости, связанные с размещением объявлений и появлением мертвых зон. И хотя для демонстрации в примерах используются объявления `let`, все то же самое относится и к `const`.

Когда движок JavaScript просматривает следующий блок кода и обнаруживает объявления переменных, он поднимает их в начало функции или глобальной области видимости (объявления `var`) или помещает в мертвую зону (объявления `let` и `const`). Любая попытка обратиться к переменной внутри мертвой зоны приводит к ошибке времени выполнения. Как только поток выполнения достигнет объявления, соответствующая переменная удаляется из мертвой зоны и становится доступной для использования.

Это происходит при любой попытке использовать переменную, объявленную с помощью `let` или `const`, до ее объявления. Как показывает предыдущий пример, ошибка возникает даже при попытке выполнить обычно безопасный оператор `typeof`. Однако может так получиться, что оператор `typeof` будет применен к переменной, объявленной во внешнем блоке, и не вызовет ошибки, но результат может оказаться не таким, какой вы ожидаете. Рассмотрим следующий фрагмент:

```
console.log(typeof value);    // "undefined"

if (condition) {
    let value = "blue";
}
```

Когда выполняется оператор `typeof`, переменная `value` не находится в мертвой зоне, потому что оператор находится за пределами блока, в котором объявляется `value`. Следовательно, в данной точке привязки `value` не существует, и `typeof` просто вернет `"undefined"`.

Временная мертвая зона — это уникальная черта, характерная для локальных привязок. Другая уникальная черта имеет отношение к использованию локальных привязок в циклах.

Блочные привязки в циклах

Одной из ситуаций, когда у разработчика возникает особенно сильное желание ограничить видимость переменной текущим блоком, является реализация циклов `for`, в которых переменная-счетчик должна использоваться только внутри цикла. Например, в программах на JavaScript часто можно встретить такой код:

```
for (var i = 0; i < 10; i++) {
    process(items[i]);
}
// переменная i все еще доступна здесь
console.log(i);    // 10
```

В других языках, где видимость переменных по умолчанию ограничивается блоком, такой код не должен работать — переменная `i` будет доступна только в цикле `for`. Но в JavaScript переменная `i` останется доступной и после завершения цикла, потому что объявление `var` будет поднято интерпретатором. Желаемое поведение дает использование объявления `let`, как показано в следующем фрагменте:

```
for (let i = 0; i < 10; i++) {
    process(items[i]);
}
// переменная i недоступна здесь - вызовет ошибку
console.log(i);
```

В этом примере переменная `i` существует только внутри цикла `for`. Когда цикл завершится, переменная окажется недоступной.

Функции в циклах

Особенности объявлений `var` долгое время усложняли определение функций в циклах, потому что переменные, объявленные таким способом внутри циклов, доступны за их пределами. Взгляните на следующий фрагмент:

```
var funcs = [];  
  
for (var i = 0; i < 10; i++) {  
    funcs.push(function() {  
        console.log(i);  
    });  
}  
funcs.forEach(function(func) {  
    func();    // десять раз выведет число "10"  
});
```

На первый взгляд кажется, что этот код должен вывести числа от 0 до 9, но в действительности он десять раз выведет число 10. Причина в том, что во всех итерациях цикла используется одна и та же переменная `i`, то есть все функции, созданные в теле цикла, будут хранить ссылку на одну и ту же переменную. После последней итерации цикла переменная `i` получит значение 10, которое и выведет все инструкции `console.log(i)` во вновь созданных функциях.

Для решения этой проблемы разработчики используют внутри циклов *выражения немедленно вызываемых функций* (*Immediately Invoked Function Expression, IIFE*), чтобы принудительно создать новую копию переменной, участвующей в итерациях, как показано в следующем примере:

```
var funcs = [];  
  
for (var i = 0; i < 10; i++) {  
    funcs.push((function(value) {  
        return function() {  
            console.log(value);  
        }  
    })(i)));  
}  
  
funcs.forEach(function(func) {  
    func();    // выведет 0, 1, 2, ..., 9  
});
```

Эта версия использует в теле цикла выражение немедленно вызываемой функции. Переменная `i` передается в выражение, которое создает собственную копию и сохраняет ее как `value`. В результате в каждой итерации сохраняется свое значение, поэтому последующие вызовы функций возвращают ожидаемые значения от 0 до 9. К счастью, блочные привязки с применением `let` и `const`, появившиеся в ECMAScript 6, могут упростить этот цикл.

Объявления `let` в циклах

Объявление `let` упрощает подобные циклы, эффективно выполняя все то, что делает выражение немедленно вызываемой функции. В каждой итерации цикл создает новую переменную и инициализирует ее значением одноименной переменной из предыдущей итерации. Благодаря этому ожидаемый результат можно получить без использования выражения немедленно вызываемой функции:

```
var funcs = [];  
  
for (let i = 0; i < 10; i++) {  
    funcs.push(function() {  
        console.log(i);  
    });  
}
```

```
}

funcs.forEach(function(func) {
    func();      // выведет 0, 1, 2, ..., 9
})
```

Этот фрагмент действует в точности как пример, использующий объявление `var` и выражение немедленно вызываемой функции, но выглядит намного проще. В каждой новой итерации объявление `let` создает новую переменную `i`, поэтому каждая функция, созданная в цикле, получает собственную копию `i`. Каждая такая копия `i` имеет свое значение, присвоенное в начале итерации. То же верно для циклов `for-in` и `for-of`, как показано ниже:

```
var funcs = [],
    object = {
        a: true,
        b: true,
        c: true
    };

for (let key in object) {
    funcs.push(function() {
        console.log(key);
    });
}

funcs.forEach(function(func) {
    func();      // выведет "a", затем "b", затем "c"
});
```

В этом примере цикл `for-in` демонстрирует то же поведение, что мы наблюдали в цикле `for`. В каждой итерации создается новая привязка `key`, поэтому каждая функция получает свою копию переменной `key`. В результате все функции выводят разные значения. Если здесь объявить `key` с помощью `var`, все функции выведут "c".

ПРИМЕЧАНИЕ

Имейте в виду, что поведение объявлений `let` в циклах отдельно определено в спецификации и, возможно, никак не связано с особенностью `let`, препятствующей «подъему». В действительности, первые реализации `let` не обладали таким поведением – оно было добавлено позднее в процессе доработки спецификации.

Объявления `const` в циклах

Спецификация ECMAScript 6 не содержит явного запрета на использование объявлений `const` в циклах; однако в разных циклах `const` действует по-разному. В обычном цикле `for` можно использовать `const` в выражении инициализации, но попытка изменить значение переменной цикла вызовет предупреждение. Например:

```
var funcs = [];

// вызовет ошибку после первой итерации
for (const i = 0; i < 10; i++) {
```



```
    funcs.push(function() {  
        console.log(i);  
    });  
}
```

Здесь переменная `i` объявлена как константа. Первая итерация, в которой `i` имеет значение 0, выполняется благополучно. Но когда подходит черед выражения `i++`, попытка изменить константу вызывает ошибку. Значит, `const` может использоваться для объявления переменной в выражении инициализации цикла, только если потом она не будет изменяться.

С другой стороны, в циклах `for-in` и `for-of` объявление `const` действует в точности как объявление `let`. Поэтому следующий фрагмент выполнится без ошибок:

```
var funcs = [],  
    object = {  
        a: true,  
        b: true,  
        c: true  
    };  
  
// не вызовет ошибку  
for (const key in object) {  
    funcs.push(function() {  
        console.log(key);  
    });  
}  
  
funcs.forEach(function(func) {  
    func();    // выведет "a", затем "b", затем "c"  
});
```

Этот код работает почти так же, как во втором примере в разделе «Объявления `let` в циклах» (см. выше). Единственное отличие — значение `key` нельзя изменить в теле цикла. Такое поведение `const` в циклах `for-in` и `for-of` объясняется тем, что в каждой итерации выражение инициализации создает новую привязку, а не пытается изменить значение существующей привязки (как это происходит в примере с циклом `for`).

Блочные привязки на глобальном уровне

Еще одно отличие `let` и `const` от `var` заключается в поведении на глобальном уровне. Когда объявление `var` используется в глобальной области видимости, оно создает новую глобальную переменную как свойство глобального объекта (`window` — в браузерах). Значит, при неосторожном обращении с `var` есть риск затереть существующее свойство, как показано ниже:

```
// в браузере  
  
var RegExp = "Hello!";  
console.log(window.RegExp);    // "Hello!"  
  
var ncz = "Hi!";  
console.log(window.ncz);    // "Hi!"
```

Несмотря на то что объект `window` уже имеет свойство `RegExp`, объявление `var` может затереть его. В этом примере объявляется новая глобальная переменная `RegExp`, затирающая оригинальное

свойство. Аналогично объявление глобальной переменной `ncz` немедленно создает свойство в объекте `window` — именно так всегда работал JavaScript.

Объявления `let` или `const` создают новые привязки в глобальной области видимости, но не добавляют свойства в глобальный объект. Это означает, что объявление `let` или `const` не затрет глобальную переменную, а просто замаскирует ее. Например:

```
// в браузере
let RegExp = "Hello!";
console.log(RegExp);           // "Hello!"
console.log(window.RegExp === RegExp); // false
const ncz = "Hi!";
console.log(ncz);              // "Hi!"
console.log("ncz" in window);  // false
```

Объявление `let RegExp` создаст привязку, которая замаскирует глобальную переменную `RegExp`. Так как `window.RegExp` и `RegExp` — не одно и то же, в глобальной области видимости не возникает беспорядка. Объявление `const ncz` также создаст привязку, но не добавит новое свойство в глобальный объект. Отсутствие влияния `let` и `const` на глобальный объект делает их использование в глобальной области видимости более безопасным, когда желательно избежать создания свойств в глобальном объекте.

ПРИМЕЧАНИЕ

Тем не менее иногда объявление `var` может оказаться востребованным, например, если имеется какой-то код, который должен быть доступен из глобального объекта. Такая потребность в основном характерна для браузеров, когда требуется обеспечить доступность кода между фреймами или окнами.

Новые приемы, появившиеся с введением блочных привязок

Пока разрабатывалась спецификация ECMAScript 6, многие полагали, что для объявления переменных вместо `var` по умолчанию должна использоваться инструкция `let`. С точки зрения многих разработчиков на JavaScript `let` действует в точности как должно было бы действовать объявление `var`, поэтому прямая замена выглядит вполне логично. А объявление `const` предполагалось использовать для защиты переменных от изменения.

Однако с увеличением числа разработчиков, мигрировавших на ECMAScript 6, получил популярность альтернативный подход — использовать `const` по умолчанию, а `let` применять, только когда известно, что значение переменной будет изменяться. Объясняется это просто: большинство переменных не должно изменять своих значений после инициализации, потому что непреднамеренное изменение — источник ошибок. Эта идея выглядит весьма заманчиво и стоит того, чтобы вы пересмотрели свой код при переходе на ECMAScript 6.

В заключение

Блочные привязки `let` и `const` вводят в JavaScript понятие лексической области видимости. Эти объявления не поднимаются интерпретатором и существуют только в блоке, где они объявлены. Блочные привязки демонстрируют поведение, более похожее на поведение переменных в других языках, и способствуют уменьшению количества непреднамеренных ошибок, потому что теперь переменные могут объявляться только там, где они необходимы. Как дополнительный эффект переменные недоступны до их объявления даже для вполне безопасных операторов, таких как `typeof`. Попытка обратиться к блочной привязке до ее объявления вызывает ошибку из-за того, что в этот момент такая привязка находится во временной мертвой зоне (TDZ).

В большинстве случаев `let` и `const` действуют, подобно `var`, однако это не относится к циклам. Внутри циклов `for-in` и `for-of` оба объявления, `let` и `const`, создают новые привязки в каждой итерации. В результате функции, созданные в теле цикла, имеют доступ к текущим значениям привязок, а не к значениям, присвоенным им в последней итерации цикла (как это характерно для `var`). То же относится к объявлениям `let` в циклах `for`, но попытка использовать в цикле `for` объявление `const` может привести к ошибке.

В настоящее время рекомендуется по умолчанию использовать объявления `const`, а `let` применять, только когда заранее известно, что значение переменной будет изменяться. Такой подход гарантирует элементарную поддержку неизменяемости переменных в коде, что может помочь избавиться от некоторых видов ошибок.

Строки и регулярные выражения

Строки, вне всяких сомнений, — один из самых важных типов данных в программировании. Они имеются практически во всех высокоуровневых языках программирования, а возможность эффективной работы с ними является для разработчиков важнейшим условием создания полезных программ. Немаловажную роль играет также поддержка регулярных выражений, потому что дает разработчикам дополнительные возможности для работы со строками. Учитывая все это, создатели ECMAScript 6 усовершенствовали поддержку строк и регулярных выражений, добавив новые и долгожданные возможности. Эта глава описывает оба типа изменений.

Улучшенная поддержка Юникода

До спецификации ECMAScript 6 строки в JavaScript были реализованы как последовательности 16-битных значений, называемых *кодowymi единицами* (*code unit*) и представляющих единственный символ. Все свойства и методы строк, такие как свойство `length` и метод `charAt()`, опирались на эти 16-битные кодовые единицы. Конечно, 16 бит было вполне достаточно, чтобы вместить любой символ. Но все изменилось с появлением расширенного набора символов, введенного стандартом Юникода.

Кодовые пункты UTF-16

Ограничение размеров символов 16 битами противоречит цели Юникода — присвоить глобально уникальный идентификатор каждому символу в мире. Эти глобально уникальные идентификаторы, называемые *кодowymi пунктами* (*code point*), — всего лишь простые числа, начиная с 0. Кодовые пункты можно считать кодами символов, то есть числами, представляющими символы. Кодировки символов определяют непротиворечивое соответствие между кодowymi пунктами и кодowymi единицами. В кодировке UTF-16 кодовые пункты могут состоять из множества кодовых единиц.

Первые 2^{16} кодовых пунктов в UTF-16 представляют отдельные 16-битные кодовые единицы. Этот диапазон называется *базовой многоязыковой плоскостью* (*Basic Multilingual Plane, BMP*). Остальное пространство занимают несколько *вспомогательных плоскостей*, в которых кодовые пункты не могут быть представлены 16 битами. В UTF-16 эта проблема решена введением *суррогатных пар*, в которых один кодовый пункт представлен двумя 16-битными кодowymi единицами. В результате каждый отдельно взятый символ в строке может быть представлен одной кодовой единицей (для символов из базовой многоязыковой плоскости), то есть 16 битами, или двумя (для символов из вспомогательных плоскостей), то есть 32 битами.

В ECMAScript 5 все строковые операции работали с 16-битными кодовыми единицами, поэтому наличие в строке суррогатных пар из кодировки UTF-16 могло приводить к неожиданным результатам, например:

```
let text = "吉";
console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(text.charAt(0));        // ""
console.log(text.charAt(1));        // ""
console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
```

Здесь единственный символ Юникода "吉" представлен суррогатными парами, в результате в данном случае строковые операции в JavaScript интерпретируют строку как состоящую из двух 16-битных символов. То есть:

- Длина текста определяется как равная 2, хотя должно быть 1.
- Регулярное выражение, предполагающее совпадение с единственным символом, терпит неудачу, потому что полагает, что в строке присутствуют два символа.
- Метод `charAt()` не в состоянии вернуть допустимую строку символов, потому что ни одна из 16-битных кодовых единиц не соответствует печатаемому символу.
- Метод `charCodeAt()` также не смог правильно идентифицировать символы. Для каждой кодовой единицы он вернул соответствующее 16-битное число, и это самое лучшее, что можно получить для фактического текста в ECMAScript 5.

Для решения подобных проблем и представления строк в ECMAScript 6 принудительно используется кодировка UTF-16. Стандартизация строковых операций на основе этой кодировки означает возможность поддержки в JavaScript операций со строками, содержащими суррогатные пары. Остальная часть этого раздела посвящена обсуждению нескольких ключевых примеров использования этих операций.

Метод `codePointAt()`

Одним из методов, добавленных в ECMAScript 6 для полноценной поддержки UTF-16, является метод `codePointAt()`, который извлекает кодовый пункт Юникода из указанной позиции в строке. Этот метод принимает позицию кодового пункта, а не символа, и возвращает целочисленное значение. Сравните эти результаты с результатами, которые возвращает `charCodeAt()`:

```
let text = "吉a";
console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
console.log(text.charCodeAt(2));    // 97

console.log(text.codePointAt(0));   // 134071
console.log(text.codePointAt(1));   // 57271
console.log(text.codePointAt(2));   // 97
```

Метод `codePointAt()` возвращает то же значение, что и метод `charCodeAt()`, только если в указанной позиции находится символ из базовой многоязыковой плоскости. Первый символ в тексте выше как раз не принадлежит этой плоскости и потому состоит из двух кодовых единиц; по этой причине свойство `length` возвращает 3, а не 2. Метод `charCodeAt()` вернул только первую кодовую единицу для позиции 0, а `codePointAt()` вернул полный кодовый пункт, несмотря на то,

что он состоит из двух кодовых единиц. Оба метода вернули одинаковые значения для позиций 1 (вторая кодовая единица первого символа) и 2 (символ "a").

Вызов метода `codePointAt()` — самый простой способ определить количество кодовых единиц, составляющих символ (1 или 2). Ниже приводится функция, которую можно было бы написать для такой проверки:

```
function is32Bit(c) {  
    return c.codePointAt(0) > 0xFFFF;  
}  
  
console.log(is32Bit("ㄱ"));    // true  
console.log(is32Bit("a"));    // false
```

Верхняя граница 16-битных числовых значений символов равна шестнадцатеричному числу `FFFF`, поэтому любые кодовые пункты, превышающие это число, состоят из двух кодовых единиц, то есть имеют размер 32 бита.

Метод `String.fromCodePoint()`

Если JavaScript поддерживает какую-то операцию, значит, он поддерживает и обратную ей операцию. Например, метод `codePointAt()` позволяет получить кодовый пункт символа в строке, а метод `String.fromCodePoint()` возвращает односимвольную строку, полученную из указанного кодового пункта:

```
console.log(String.fromCodePoint(134071));    // ㄱ
```

Метод `String.fromCodePoint()` можно считать более полной версией метода `String.fromCharCode()`. Оба возвращают одинаковые результаты для всех кодовых пунктов из базовой многоязыковой плоскости. И только когда им передаются кодовые пункты из вспомогательных плоскостей, их результаты отличаются.

Метод `normalize()`

Еще один интересный аспект Юникода — возможность считать два разных символа эквивалентными при выполнении сортировки и других операций, основанных на сравнении. Существует два вида подобных отношений между символами. Первый — *каноническая эквивалентность*, когда две последовательности кодовых пунктов являются полностью взаимозаменяемыми. Например, комбинация двух символов может быть канонически эквивалентна одному символу. Второй вид отношений — *совместимость*. Две совместимые последовательности кодовых пунктов выглядят по-разному, но в некоторых ситуациях могут быть взаимозаменяемыми.

Из-за этих отношений две строки, представляющие по сути один и тот же текст, могут содержать разные последовательности кодовых пунктов. Например, символ «ж» и двухсимвольную строку «ae» можно использовать взаимозаменяемо, но они не являются строго эквивалентными, если не провести некоторую нормализацию.

ECMAScript 6 поддерживает несколько форм нормализации строк Юникода с помощью метода `normalize()`. В этот метод может передаваться дополнительный однострочный параметр, определяющий форму нормализации:

- Форма нормализации канонической композицией (Normalization Form Canonical Composition, "NFC"), по умолчанию.
- Форма нормализации канонической декомпозицией (Normalization Form Canonical Decomposition, "NFD").

- Форма нормализации совместимой композицией (Normalization Form Compatibility Composition, "NFKC").
- Форма нормализации совместимой декомпозицией (Normalization Form Compatibility Decomposition, "NFKD").

Разъяснение различий между этими четырьмя формами выходит за рамки данной книги. Поэтому просто запомните, что перед сравнением двух строк их необходимо нормализовать к одной и той же форме. Например:

```
let normalized = values.map(function(text) {
  return text.normalize();
});

normalized.sort(function(first, second) {
  if (first < second) {
    return -1;
  } else if (first === second) {
    return 0;
  } else {
    return 1;
  }
});
```

Этот код преобразует строки, хранящиеся в массиве `values`, в нормализованную форму, чтобы получить возможность отсортировать данный массив. Аналогично можно отсортировать исходный массив, вызывая `normalize()` в функции сравнения, как показано ниже:

```
values.sort(function(first, second) {
  let firstNormalized = first.normalize(),
      secondNormalized = second.normalize();

  if (firstNormalized < secondNormalized) {
    return -1;
  } else if (firstNormalized === secondNormalized) {
    return 0;
  } else {
    return 1;
  }
});
```

И снова, самое важное в этом примере, на что следует обратить внимание, — обе строки, `first` и `second`, нормализуются к одной форме. В этих примерах по умолчанию используется форма NFC, но точно так же можно использовать любую из поддерживаемых форм:

```
values.sort(function(first, second) {
  let firstNormalized = first.normalize("NFD"),
      secondNormalized = second.normalize("NFD");

  if (firstNormalized < secondNormalized) {
    return -1;
  } else if (firstNormalized === secondNormalized) {
    return 0;
  } else {
    return 1;
  }
});
```

```
        return 1;
    }
});
```

Если прежде у вас не было причин задумываться о нормализации, вполне возможно, что вы и впредь сможете обходиться без этого метода. Но если вам доводилось разрабатывать интернационализированные приложения, метод `normalize()` без сомнений пригодится.

Новые методы — не единственное усовершенствование, улучшающее поддержку строк Юникода. Спецификация ECMAScript 6 также вводит флаг `u` в регулярные выражения и определяет другие изменения, касающиеся строк и регулярных выражений.

Флаг `u` в регулярных выражениях

Многие типовые операции со строками можно реализовать с применением регулярных выражений. Но регулярные выражения оперируют символами, представленными 16-битными кодовыми единицами. Чтобы решить эту проблему, ECMAScript 6 определяет флаг `u` (от англ. *Unicode* — Юникод) для использования в регулярных выражениях.

Действие флага `u`

Если в регулярном выражении установлен флаг `u`, оно переключается в режим работы с символами, а не с кодовыми единицами. Это означает, что регулярное выражение не будет интерпретировать суррогатные пары как два отдельных символа и должно действовать, как ожидается. Например:

```
let text = "𐄂";

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(/^.$/u.test(text));     // true
```

Регулярное выражение `/^.$/` соответствует любой строке, содержащей единственный символ. Без флага `u` это регулярное выражение проверяет соответствие с кодовыми единицами, поэтому японский символ (представленный двумя кодовыми единицами) не совпадает с данным регулярным выражением. С установленным флагом `u` регулярное выражение проверяет соответствие с символами, а не с кодовыми единицами, благодаря чему обнаруживается его совпадение с единственным японским символом.

Подсчет кодовых пунктов

К сожалению, спецификация ECMAScript 6 не определяет метод для подсчета количества кодовых пунктов в строке (свойство `length` по-прежнему возвращает количество кодовых единиц), однако для этой цели можно использовать регулярное выражение с флагом `u`:

```
function codePointLength(text) {
    let result = text.match(/[\s\S]/gu);
    return result ? result.length : 0;
}

console.log(codePointLength("abc")); // 3
console.log(codePointLength("𐄂bc")); // 3
```


В этом примере вызывается метод `match()`, чтобы проверить `text` на наличие пробельных и непробельных символов (шаблон `[\s\S]` гарантирует совпадение с символами перевода строки) с помощью регулярного выражения, в которое включены глобальный режим поиска и поддержка Юникода. Возвращаемый результат содержит массив совпадений, если имеется хотя бы одно совпадение, то есть длина массива совпадает с количеством кодовых пунктов в строке. Строки Юникода `"abc"` и `"ㄱbc"` имеют по три символа, поэтому длина массива в обоих случаях равна трем.

ПРИМЕЧАНИЕ

Несмотря на то что это решение вполне работоспособно, оно не отличается высокой скоростью, особенно при анализе длинных строк. Более быстрое решение основывается на применении строкового итератора (обсуждается в главе 8). А вообще, старайтесь минимизировать количество операций подсчета кодовых пунктов.

Определение поддержки флага `u`

Поскольку флаг `u` относится к изменениям в синтаксисе, попытка использовать его в движках JavaScript, не совместимых с ECMAScript 6, вызовет синтаксическую ошибку. Поддерживается ли флаг `u`, можно безопасно определить с помощью следующей функции:

```
function hasRegExpU() {
    try {
        var pattern = new RegExp(".", "u");
        return true;
    } catch (ex) {
        return false;
    }
}
```

Эта функция вызывает конструктор `RegExp` и передает ему флаг `u` в качестве аргумента. Такой синтаксис допустим даже при работе с более ранними версиями JavaScript, но конструктор вызовет ошибку, если флаг `u` не поддерживается.

ПРИМЕЧАНИЕ

Если ваш код должен сохранять работоспособность при выполнении старыми движками JavaScript, всегда используйте конструктор `RegExp` для проверки поддержки флага `u`. Это поможет предотвратить появление синтаксических ошибок и позволит выяснить возможность применения флага `u`, не вызывая остановку выполнения программы.

Другие изменения в поддержке строк

Поддержка операций со строками в JavaScript всегда отставала от аналогичной поддержки в других языках. Например, метод `trim()` появился только в ECMAScript 5, и ECMAScript 6 продолжила расширение поддержки строк в JavaScript.

Методы идентификации подстрок

Начиная с первых версий JavaScript, для идентификации подстрок в строках разработчики используют метод `indexOf()`, и они давно просят дать им более простой способ. В результате в ECMAScript 6 были добавлены три новых метода:

- Метод `includes()` возвращает `true`, если указанный текст присутствует в строке. В противном случае возвращается `false`.
- Метод `startsWith()` возвращает `true`, если указанный текст присутствует в начале строки. В противном случае возвращается `false`.
- Метод `endsWith()` возвращает `true`, если указанный текст присутствует в конце строки. В противном случае возвращается `false`.

Все три метода принимают два аргумента: искомый текст и необязательный индекс, с которого начинается поиск. При наличии второго аргумента `includes()` и `startsWith()` начинают поиск с указанного индекса, а `endsWith()` начинает поиск с конца строки, отступив к началу на величину второго аргумента; если второй аргумент отсутствует, `includes()` и `startsWith()` выполняют поиск с начала строки, а `endsWith()` — с конца. Фактически второй аргумент помогает сократить область поиска. Ниже демонстрируется несколько примеров применения этих трех методов:

```
let msg = "Hello world!";

console.log(msg.startsWith("Hello"));    // true
console.log(msg.endsWith("|"));          // true
console.log(msg.includes("o"));          // true

console.log(msg.startsWith("o"));        // false
console.log(msg.endsWith("world!"));     // true
console.log(msg.includes("x"));          // false

console.log(msg.startsWith("o", 4));     // true
console.log(msg.endsWith("o", 8));       // true
console.log(msg.includes("o", 8));       // false
```

Первые три вызова осуществляются без второго параметра, поэтому они выполняют поиск по всей строке, если потребуется. Последние три вызова выполняют поиск в ограниченном фрагменте строки. Вызов `msg.startsWith("o 4)` начинает поиск с 4-й позиции в строке `msg`, где находится символ *o* в слове *Hello*. Вызов `msg.endsWith("o 8)` также начинает поиск с 4-й позиции в строке, потому что при вычитании аргумента 8 из длины строки (12) получается число 4. Вызов `msg.includes("o 8)` начинает поиск с 8-й позиции, где находится символ *r* в слове *world*.

Эти три метода действительно упрощают поиск подстроки в строке, но все они возвращают логическое значение. Если вам понадобится найти фактическую позицию одной строки в другой, используйте метод `indexOf()` или `lastIndexOf()`.

ПРИМЕЧАНИЕ

Методы `startsWith()`, `endsWith()` и `includes()` вызывают ошибку, если вместо строки передать регулярное выражение. Методы `indexOf()` и `lastIndexOf()`, напротив, преобразуют аргумент с регулярным выражением в строку и выполняют поиск в этой строке.

Метод `repeat()`

Спецификация ECMAScript 6 добавила в поддержку строк еще один новый метод — метод `repeat()`, который принимает аргумент с количеством повторений строки. Он возвращает новую строку, содержащую исходную указанное количество раз. Например:

```
console.log("x".repeat(3));      // "xxx"
console.log("hello".repeat(2));  // "hellohello"
console.log("abc".repeat(4));    // "abcabcabcabc"
```

Этот метод был добавлен в основном для удобства манипулирования текстом. В частности, он может пригодиться в реализациях утилит форматирования, которые должны создавать отступы в зависимости от уровня вложенности, например:

```
// отступ с определенным количеством пробелов
let indent = " ".repeat(4),
    indentLevel = 0;

// всякий раз, когда нужно увеличить отступ
let newIndent = indent.repeat(++indentLevel);
```

Первый вызов `repeat()` создает строку с четырьмя пробелами, а переменная `indentLevel` хранит уровень вложенности (число отступов). После этого можно просто вызывать `repeat()` с увеличенным значением `indentLevel`, чтобы изменить величину отступа.

ECMAScript 6 также вносит некоторые полезные изменения в регулярные выражения, которые не относятся к какой-либо конкретной категории. Некоторые из них описываются в следующем разделе.

Другие изменения в регулярных выражениях

Регулярные выражения — важная часть арсенала инструментов для работы со строками в JavaScript, и, подобно многим другим инструментам языка, они мало менялись в последних версиях. Однако в ECMAScript 6 было внесено несколько усовершенствований в регулярные выражения, дополняющих нововведения в поддержке строк.

Флаг `y` в регулярных выражениях

Спецификация ECMAScript 6 стандартизовала флаг `y` после его реализации в Firefox в виде проприетарного расширения регулярных выражений. Флаг `y` воздействует на свойство `sticky` регулярных выражений и требует, чтобы поиск совпадений в строке начинался с позиции, указанной в свойстве `lastIndex` регулярного выражения. Если совпадение в данной позиции не обнаружено, регулярное выражение прекращает поиск. Следующий пример показывает, как это работает:

```
let text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);      // "hello1 "
console.log(globalResult[0]); // "hello1 "
console.log(stickyResult[0]); // "hello1 "

pattern.lastIndex = 1;
globalPattern.lastIndex = 1;
stickyPattern.lastIndex = 1;
```

```
result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);           // "hello1 "
console.log(globalResult[0]);     // "hello2 "
console.log(stickyResult[0]);     // вызовет ошибку!
```

В этом примере используется три регулярных выражения. Выражение в переменной `pattern` не имеет флагов, в выражении `globalPattern` установлен флаг `g` и в выражении `stickyPattern` установлен флаг `y`. Первая тройка вызовов `console.log()` для всех трех регулярных выражений выводит "hello1" с пробелом в конце.

Затем свойству `lastIndex` всех трех шаблонов присваивается значение 1, требующее начинать поиск совпадения с регулярным выражением со второго символа. Регулярное выражение без флагов полностью игнорирует свойство `lastIndex` и благополучно находит совпадение с "hello1". Регулярное выражение с флагом `g` находит совпадение с "hello2", потому что начинает поиск со второго символа в строке ("e"). Регулярное выражение `stickyPattern` не обнаруживает совпадения, начиная со второго символа, поэтому `stickyResult` получает значение `null`.

Флаг `y` сохраняет индекс символа, следующего за последним совпадением, в свойстве `lastIndex` после каждой операции. Если совпадения не нашлось, в `lastIndex` записывается начальное значение 0. Аналогично действует флаг глобального поиска, как показано ниже:

```
let text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);           // "hello1 "
console.log(globalResult[0]);     // "hello1 "
console.log(stickyResult[0]);     // "hello1 "

console.log(pattern.lastIndex);   // 0
console.log(globalPattern.lastIndex); // 7
console.log(stickyPattern.lastIndex); // 7

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);           // "hello1 "
console.log(globalResult[0]);     // "hello2 "
console.log(stickyResult[0]);     // "hello2 "

console.log(pattern.lastIndex);   // 0
console.log(globalPattern.lastIndex); // 14
console.log(stickyPattern.lastIndex); // 14
```

В обеих переменных, `stickyPattern` и `globalPattern`, свойство `lastIndex` получило значение 7 после первого вызова `exec()` и значение 14 после второго вызова.

Запомните две важные особенности флага `y`. Во-первых, значение свойства `lastIndex` учитывается только при вызове методов существующего объекта регулярного выражения, таких как `exec()` и `test()`. Передача флага `y` методам строк, таким как `match()`, не оказывает никакого влияния.

Во-вторых, когда в регулярном выражении с флагом `y` используется символ `^`, соответствующий началу строки, совпадение возможно только с начала строки (или с первого символа, следующего за символом перевода строки, если поиск выполняется в многострочном режиме). Если свойство `lastIndex` будет иметь значение 0, регулярное выражение с символом `^` будет действовать, как выражение без флага `y`. Если значение свойства `lastIndex` не будет соответствовать началу строки в режиме однострочного поиска или началу одной из строк в режиме многострочного поиска, регулярное выражение с флагом `y` никогда не найдет совпадения.

Как и в случае с другими флагами регулярных выражений, определить присутствие флага `y` можно с помощью свойства. В данном случае достаточно проверить наличие свойства `sticky`:

```
let pattern = /hello\d/y;

console.log(pattern.sticky);    // true
```

Свойство `sticky` получает значение `true`, если флаг `y` присутствует в регулярном выражении, и `false` — если отсутствует. Это свойство доступно только для чтения и не может изменяться непосредственно.

Подобно флагу `u`, флаг `y` относится к изменениям в синтаксисе, поэтому его применение может вызывать синтаксическую ошибку в движках JavaScript, не совместимых с ECMAScript 6. Безопасно определить, поддерживается ли флаг `y`, можно с помощью следующей функции:

```
function hasRegExpY() {
    try {
        var pattern = new RegExp(".", "y");
        return true;
    } catch (ex) {
        return false;
    }
}
```

Подобно проверке поддержки флага `u`, эта функция возвращает `false`, если ей не удалось создать регулярное выражение с флагом `y`. Кроме того, если ваш код должен сохранять работоспособность при выполнении старыми движками JavaScript, всегда используйте конструктор `RegExp` для определения таких регулярных выражений, чтобы избежать синтаксических ошибок.

Создание копий регулярных выражений

В ECMAScript 5 имелась возможность создавать копии регулярных выражений, передавая их в конструктор `RegExp`, например:

```
var re1 = /ab/i,
    re2 = new RegExp(re1);
```

Переменная `re2` — это всего лишь копия переменной `re1`. Но если в вызов конструктора `RegExp` передать второй аргумент, определяющий флаги для регулярного выражения, такой код не будет работать, например:

```
var re1 = /ab/i,

// вызовет ошибку в ES5, выполнится без ошибки в ES6
re2 = new RegExp(re1, "g");
```

Если выполнить этот код в окружении ECMAScript 5, он вызовет ошибку с сообщением о том, что второй аргумент не может использоваться, когда первый является регулярным выражением. В ECMAScript 6 это поведение было изменено, и теперь допускается передавать второй аргумент, переопределяющий флаги, присутствующие в первом аргументе. Например:

```
let re1 = /ab/i,

    // вызовет ошибку в ES5, выполнится без ошибки в ES6
    re2 = new RegExp(re1, "g");

console.log(re1.toString()); // "/ab/i"
console.log(re2.toString()); // "/ab/g"

console.log(re1.test("ab")); // true
console.log(re2.test("ab")); // true

console.log(re1.test("AB")); // true
console.log(re2.test("AB")); // false
```

В этом примере `re1` имеет флаг `i` (нечувствительность к регистру символов), а `re2` имеет только флаг `g` (глобальный поиск). Конструктор `RegExp` скопирует шаблон из `re1` и подставит флаг `g` вместо флага `i`. При вызове конструктора без второго аргумента `re2` получит те же флаги, что и `re1`.

Свойство `flags`

Помимо нового флага и возможности изменять имеющиеся флаги спецификация ECMAScript 6 добавила новое свойство, связанное с ними. В ECMAScript 5 есть возможность получить текст регулярного выражения с помощью свойства `source`, но чтобы получить строку с флагами, требуется выполнить парсинг результата вызова метода `toString()`, как показано ниже:

```
function getFlags(re) {
    var text = re.toString();
    return text.substring(text.lastIndexOf("/") + 1, text.length);
}

// toString() вернет "/ab/g"
var re = /ab/g;

console.log(getFlags(re)); // "g"
```

Этот фрагмент преобразует регулярное выражение в строку и возвращает символы, найденные после последнего символа `/`. Эти символы обозначают флаги.

ECMAScript 6 упрощает получение флагов, добавляя свойство `flags` в пару к свойству `source`. Оба свойства являются свойствами прототипа и доступны только для чтения. Свойство `flags` упрощает исследование регулярных выражений при отладке и наследовании.

Последнее нововведение в ECMAScript 6 — свойство `flags` — возвращает строковое представление флагов, присутствующих в регулярном выражении. Например:

```
let re = /ab/g;

console.log(re.source); // "ab"
console.log(re.flags);  // "g"
```

Чтобы извлечь из `re` все флаги и вывести их в консоль, в этом примере потребовалось намного меньше строк кода, чем в примере, демонстрирующем прием с использованием `toString()`. Свойства `source` и `flags` позволяют извлекать фрагменты регулярного выражения без необходимости выполнять парсинг строки с регулярным выражением.

Изменения в поддержке строк и регулярных выражений, до сих пор обсуждавшиеся в этой главе, определенно расширяют ваши возможности, но ECMAScript 6 вводит еще более существенные усовершенствования в поддержку строк. Одним из них является новый тип строковых литералов, делающий строки еще более гибкими.

Литералы шаблонов

Чтобы дать разработчикам возможность решать более сложные задачи, синтаксис *литералов шаблонов* в ECMAScript 6 позволяет определять предметно-ориентированные языки (Domain-Specific Languages, DSL) для более безопасной работы с содержимым, чем это позволяют решения, доступные в ECMAScript 5 и более ранних версиях. Предметно-ориентированный язык — это язык программирования, предназначенный для решения в узкой, специализированной области, в противоположность универсальным языкам, таким как JavaScript. На вики-странице ECMAScript (<http://wiki.ecmascript.org/doku.php?id=harmony:quasis/>) предлагается следующее черновое описание литералов шаблонов:

Эта схема расширяет синтаксис ECMAScript синтаксическим сахаром, позволяющим разработчикам библиотек определять свои предметно-ориентированные языки для создания и извлечения содержимого и выполнения операций с ним, невосприимчивые или устойчивые к атакам внедрения, таким как XSS, внедрение SQL и др.

Но в действительности литералы шаблонов — это ответ ECMAScript 6 на потребность в следующих возможностях JavaScript, отсутствовавших в версиях ECMAScript 5 и ниже:

- **Литералы многострочного текста.** Воплощение идеи поддержки многострочных строк.
- **Простое форматирование строк.** Возможность подстановки в строки значений переменных.
- **Экранирование HTML.** Возможность преобразовать строку так, чтобы ее можно было безопасно вставить в разметку HTML.

Вместо расширения функциональных возможностей строк JavaScript литералы шаблонов открывают совершенно новый подход к решению перечисленных проблем.

Основной синтаксис

В простейшем виде литералы шаблонов действуют подобно обычным строкам, отличаясь только ограничивающими символами — обратные апострофы (```) вместо двойных или одиночных кавычек. Например:

```
let message = `Hello world!`;

console.log(message);           // "Hello world!"
console.log(typeof message);    // "string"
console.log(message.length);    // 12
```

Этот пример показывает, что переменная `message` хранит обычную для JavaScript строку. Здесь с помощью синтаксиса литералов шаблонов было создано строковое значение, которое затем было присвоено переменной `message`.

Если потребуется использовать обратные апострофы в строке, просто экранируйте их символами обратного слеша (\), как в следующей версии переменной `message`:

```
let message = ``Hello\` world!`;

console.log(message);           // "`Hello` world!"
console.log(typeof message);    // "string"
console.log(message.length);    // 14
```

В литералах шаблонов не требуется экранировать двойные или одиночные кавычки.

Многострочный текст

Разработчики на JavaScript еще с самых первых версий языка ищут способы определения многострочного текста. Но строки в двойных или одиночных кавычках требуют, чтобы весь текст был определен в одной строке кода.

Обходные решения до появления ECMAScript 6

Благодаря древней синтаксической ошибке в JavaScript имеется обходной путь для создания строк с многострочным текстом: достаточно добавить символ обратного слеша (\) перед символами перевода строки. Например:

```
var message = "Multiline \
string";

console.log(message);    // "Multiline string"
```

При выводе в консоль обнаруживается, что символ перевода строки отсутствует в строке `message`. Это объясняется тем, что символ обратного слеша интерпретируется как признак продолжения, а не как перевод строки.

Чтобы вывести символ перевода строки, его нужно добавить вручную:

```
var message = "Multiline \n\
string";

console.log(message);    // "Multiline
                        // string"
```

Этот код должен вывести содержимое `message` в двух отдельных строках во всех основных движках JavaScript; однако такое поведение объявлено ошибочным, и многие разработчики рекомендуют не использовать данный прием.

Другие приемы создания строк с многострочным текстом, использовавшиеся до появления ECMAScript 6, обычно основаны на применении массивов или конкатенации строк, как в следующем примере:

```
var message = [
    "Multiline ",
    "string"
].join("\n");

let message = "Multiline \n" +
    "string";
```

Но любые обходные решения, использовавшиеся разработчиками для определения многострочного текста, были не очень практичны и неудобны.

Многострочный текст — простой путь

Литералы шаблонов в ECMAScript 6 делают объявление многострочного текста простым благодаря отсутствию специального синтаксиса. Просто включайте символы перевода строки в нужные места, и они появятся в результате, например:

```
let message = `Multiline
string`;

console.log(message);           // "Multiline
                                // string"
console.log(message.length);    // 16
```

Все пробельные символы внутри обратных апострофов становятся частью строки, поэтому будьте внимательнее, оформляя отступы. Например:

```
let message = `Multiline
                string`;
console.log(message);           // "Multiline
                                //
                                string"
console.log(message.length);    // 31
```

В этом примере все пробельные символы в начале второй строки в литерале шаблона вошли в состав получившегося строкового значения.

Если выравнивание и отступы играют важную роль, оставьте пустой первую строку в литерале многострочного шаблона и затем оформляйте отступы как требуется, например:

```
let html = `
<div>
  <h1>Title</h1>
</div>`.trim();
```

В этом фрагменте литерал шаблона начинается с пустой строки. В последующих строках теги HTML размещены с отступами, чтобы придать разметке структурный вид, а вызов метода `trim()` в конце удаляет первую пустую строку.

При желании в литералах шаблонов можно использовать управляющую комбинацию `\n` вместо символа перевода строки:

```
let message = `Multiline\nstring`;

console.log(message);           // "Multiline
                                // string"
console.log(message.length);    // 16
```

Подстановка значений

Сейчас литералы шаблонов кому-то могут показаться всего лишь непривычной версией обычных строк. Истинное отличие между ними заключается в возможности *подстановки значений*, позволяющей встраивать в литералы шаблонов допустимые выражения на JavaScript и получать на выходе строку с внедренными результатами вычисления этих выражений.

Встраиваемые выражения в литералах начинаются парой символов `${` и завершаются `}`. Простейшим примером подстановки может служить внедрение локальных переменных, например:

```
let name = "Nicholas",
    message = `Hello, ${name}.`;

console.log(message);    // "Hello, Nicholas."
```

Подстановка `${name}` извлекает значение локальной переменной `name` и вставляет его в строку `message`. В результате переменная `message` немедленно получает результат подстановки.

ПРИМЕЧАНИЕ

Литералы шаблонов имеют доступ ко всем переменным в области видимости, где они (литералы) определены. Попытка использовать необъявленную переменную вызывает ошибку в обоих режимах выполнения, строгом и нестрогом.

Так как все подстановки являются выражениями на языке JavaScript, вы можете внедрять более сложные конструкции, чем просто имена переменных. При необходимости можно встраивать арифметические выражения, вызовы функций и многое другое. Например:

```
let count = 10,
    price = 0.25,
    message = `${count} items cost $${(count * price).toFixed(2)}.`;

console.log(message);    // "10 items cost $2.50."
```

В этом примере в литерал шаблона встроено арифметическое выражение. Это выражение находит произведение переменных `count` и `price`, которое затем форматируется вызовом `.toFixed()` для вывода двух знаков после десятичной точки. Знак доллара перед второй подстановкой соответствует самому себе и выводится как знак доллара, потому что за ним не следует открывающая фигурная скобка.

Литералы шаблонов сами являются выражениями JavaScript, поэтому их можно включать в другие литералы шаблонов, как в следующем примере:

```
let name = "Nicholas",
    message = `Hello, ${
        `my name is ${ name }`
    }.`;

console.log(message);    // "Hello, my name is Nicholas."
```

В этом примере один литерал шаблона вложен в другой. За первой парой символов `${` следует второй литерал шаблона. Вторая пара `${` отмечает начало встроеного выражения внутри вложенного литерала шаблона. Этим выражением является имя переменной `name`, значение которой вставляется в результат.

Теги шаблонов

Теперь вы знаете, как с помощью литералов шаблонов определять многострочный текст и вставлять в строки значения, не используя операцию конкатенации. Но истинная их мощь заключается в возможности определять теги шаблонов. *Тег шаблона (template tag)* преобразует литерал шаблона и возвращает получившееся строковое значение. Имя тега определяется в начале шаблона — непосредственно перед первым символом ```, как показано ниже:

```
let message = tag`Hello world`;
```

В данном примере `tag` — это имя тега шаблона, при обращении к которому применяется литерал шаблона ``Hello world``.

Определение тегов

Тег — это всего лишь функция, которая вызывается для обработки данных литерала шаблона. Она принимает информацию о шаблоне в виде отдельных фрагментов, которые требуется объединить, чтобы получить результат. Первый аргумент — массив литеральных строк, полученных движком JavaScript из исходного кода. Все остальные аргументы — интерпретированные значения для подстановки.

Функции тегов обычно определяются с применением остаточных аргументов (rest arguments), чтобы упростить обработку данных и избавиться от использования именованных параметров, как показано ниже:

```
function tag(literals, ...substitutions) {  
    // вернуть строку  
}
```

Чтобы лучше понять, что получают функции тегов, рассмотрим следующий фрагмент:

```
let count = 10,  
    price = 0.25,  
    message = passthru`${count} items cost $$${(count * price).toFixed(2)}.`;
```

Гипотетическая функция `passthru()`, присутствующая в данном примере, получит два аргумента при использовании ее в качестве тега литерала шаблона. Первый аргумент — массив литералов `literals` со следующими элементами:

- Пустая строка перед первым выражением подстановки (`""`).
- Строка после первого выражения подстановки и перед вторым (`"items cost $"`).
- Строка после второго выражения подстановки (`"."`).

Следующий аргумент — число 10, получающееся в результате интерпретации переменной `count`. Это значение станет первым элементом в массиве `substitutions`. Третий аргумент — строка `"2.50"`, полученная в результате интерпретации выражения `(count * price).toFixed(2)`. Он станет вторым элементом массива `substitutions`.

Обратите внимание, что первый элемент в массиве `literals` является пустой строкой. Такое решение гарантирует, что `literals[0]` всегда соответствует началу строки результата, так же как `literals[literals.length - 1]` всегда соответствует концу строки. Количество элементов в массиве `substitutions` всегда на единицу меньше количества элементов в массиве `literals`, то есть выражение `substitutions.length === literals.length - 1` всегда истинно.

Благодаря такой организации аргументов можно легко объединить массивы `literals` и `substitutions`, чтобы получить строку результата. Сначала в строку помещается первый элемент массива `literals`, затем первый элемент массива `substitutions` и так далее до тех пор, пока строка не будет сконструирована. Для примера попробуем симитировать поведение по умолчанию литерала шаблона, чередуя значения элементов из двух массивов:

```
function passthru(literals, ...substitutions) {  
    let result = "";  
  
    // цикл по количеству элементов в массиве substitution  
    for (let i = 0; i < substitutions.length; i++) {  
        result += literals[i];  
        result += substitutions[i];  
    }  
}
```

```
// добавить последний литерал
result += literals[literals.length - 1];

return result;
}

let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $$${(count * price).toFixed(2)}.`;

console.log(message);    // "10 items cost $2.50."
```

В этом примере определяется тег `passthru`, реализующий поведение по умолчанию литерала шаблона. Единственная хитрость — использование `substitutions.length` для ограничения количества итераций вместо `literals.length`, чтобы исключить вероятность выхода за границы массива `substitutions`. Этот трюк работает, потому что отношения между `literals` и `substitutions` четко определены в спецификации ECMAScript 6.

ПРИМЕЧАНИЕ

Значения элементов в массиве `substitutions` не всегда являются строками. Если выражение возвращает число, как в предыдущем примере, в массив записывается числовое значение. Определение способа вывода таких значений как раз и является задачей тега.

Использование необработанных значений

Теги шаблонов также имеют доступ к исходной информации о строках, которая используется в основном для доступа к экранированным последовательностям, имевшим место до их преобразования в символьные эквиваленты. В простейшем случае для работы с необработанными строковыми значениями используется встроенный тег `String.raw()`. Например:

```
let message1 = `Multiline\nstring`,
    message2 = String.raw`Multiline\nstring`;
console.log(message1);    // "Multiline
                           // string"
console.log(message2);    // "Multiline\\nstring"
```

В этом фрагменте последовательность `\n` в объявлении переменной `message1` интерпретируется как символ перевода строки, а в объявлении переменной `message2` она возвращается в необработанной форме `"\\n"` (символы слеша и `n`). Возможность получить необработанную строку, как в данном примере, позволяет организовать более сложную обработку.

Необработанные строки также передаются в теги шаблонов. Первый аргумент функции тега — массив — имеет дополнительное свойство `raw`, которое хранит массив с необработанными эквивалентами литеральных значений. Например, для значения `literals[0]` всегда имеется необработанный эквивалент в `literals.raw[0]`. Зная это, можно симитировать поведение `String.raw()`, как показано ниже:

```
function raw(literals, ...substitutions) {
    let result = "";

    // цикл по количеству элементов в массиве substitution
    for (let i = 0; i < substitutions.length; i++) {
        // использовать необработанные значения
```

```
        result += literals.raw[i];
        result += substitutions[i];
    }

    // добавить последний литерал
    result += literals.raw[literals.length - 1];

    return result;
}

let message = raw`Multiline\nstring`;

console.log(message);           // "Multiline\nstring"
console.log(message.length);    // 17
```

В этом примере для получения строки с результатом используется `literals.raw` вместо `literals`. Это означает, что любые экранированные последовательности, включая последовательности, соответствующие кодовым пунктам Юникода, будут возвращены в их исходной, необработанной форме. Необработанные строки могут пригодиться для вывода строк, содержащих код, включающий экранированные последовательности. Например, если потребуется сгенерировать документацию с описанием некоторого кода, вы сможете вывести фактический код в его первоначальном виде.

В заключение

Полноценная поддержка Юникода в ECMAScript 6 позволяет сценариям на JavaScript обрабатывать строки в кодировке UTF-16 логичными способами. Появление возможности преобразований между кодовыми пунктами и символами с помощью `codePointAt()` и `String.fromCodePoint()` — важное усовершенствование для работы со строками. Добавление флага `u` в регулярные выражения открывает возможность оперировать не только 16-битными символами, но и кодовыми пунктами, а появление метода `normalize()` дает возможность сравнивать эквивалентные строки.

Спецификация ECMAScript 6 добавила новые методы для работы со строками, упрощающие поиск подстрок независимо от их местоположения в строке. В регулярные выражения также были добавлены дополнительные возможности.

Литералы шаблонов — важное дополнение, появившееся в ECMAScript 6, которое позволяет создавать предметно-ориентированные языки (Domain-Specific Languages, DSL) с целью упростить конструирование строк. Возможность внедрять переменные непосредственно в литералы шаблонов дает разработчикам более безопасный способ создания строк, чем операция конкатенации.

Встроенная поддержка определения многострочного текста также оказывается ценным усовершенствованием в сравнении с обычными строками JavaScript, которые никогда не давали такой возможности. Несмотря на то что литералы шаблонов допускают непосредственное использование символов перевода строки, вы все еще можете использовать `\n` и другие экранированные последовательности.

Теги шаблонов — наиболее важная часть поддержки литералов шаблонов, способствующая созданию предметно-ориентированных языков. Теги — это функции, получающие литералы шаблонов, разбитые на части. Эти данные можно использовать для конструирования возвращаемой строки. Данные, доступные в функции тега, включают массив литералов, их необработанные эквиваленты и значения для подстановки. Эти данные помогают правильно сформировать результат тега.

Функции — важный элемент любого языка программирования, и до появления ECMAScript 6 функции в JavaScript почти не претерпели изменений со времен появления языка. В результате накопилось множество старых проблем и нюансов поведения, которые способствовали появлению ошибок и часто требовали писать больше кода, чтобы получить самое простое поведение.

Спецификация ECMAScript 6 сделала большой шаг вперед — в ней учтены жалобы и предложения разработчиков на JavaScript, копившиеся годами. В результате появилось множество усовершенствований, расширяющих функции ECMAScript 5, которые повышают гибкость программирования на JavaScript и уменьшают вероятность ошибок.

Функции со значениями параметров по умолчанию

Функции в JavaScript уникальны тем, что позволяют передавать им любое число параметров независимо от числа параметров, объявленных в определении функции. Это дает возможность писать функции, обрабатывающие произвольное число параметров, часто подставляя значения по умолчанию вместо параметров, которые не были переданы при вызове. В этом разделе рассказывается, как действует механизм параметров по умолчанию до и после появления ECMAScript 6, а также дается некоторая важная информация об объекте `arguments`, использовании выражений в качестве параметров и временной мертвой зоне (TDZ).

Имитация значений параметров по умолчанию в ECMAScript 5

В ECMAScript 5 и более ранних версиях обычно приходилось использовать следующий прием, чтобы определить функцию со значениями параметров по умолчанию:

```
function makeRequest(url, timeout, callback) {  
  
    timeout = timeout || 2000;  
    callback = callback || function() {};  
  
    // остальная часть функции  
}
```

В этом примере параметры `timeout` и `callback` являются необязательными, потому что получают значения по умолчанию, если они не указаны при вызове. Оператор логического ИЛИ (`||`)

всегда возвращает второй операнд, если первый имеет ложное значение. Поскольку именованные параметры, которые не были явно переданы в вызов функции, получают значение `undefined`, оператор логического ИЛИ часто применяется для присваивания значений по умолчанию отсутствующим параметрам. Однако этот прием имеет существенный недостаток — в параметре `timeout` может быть передано вполне допустимое значение 0, которое будет заменено значением 2000, потому что 0 считается ложным значением.

В подобных случаях предпочтительнее использовать более безопасную альтернативу, основанную на проверке типа аргумента с помощью `typeof`, как в следующем примере:

```
function makeRequest(url, timeout, callback) {

    timeout = (typeof timeout !== "undefined") ? timeout : 2000;
    callback = (typeof callback !== "undefined") ? callback : function() {};

    // остальная часть функции
}
```

Хотя это решение более безопасное, оно по-прежнему вынуждает писать дополнительный код, чтобы выполнить элементарную операцию. Данный подход представляет типичный шаблон программирования, и многие популярные библиотеки на JavaScript переполнены подобными шаблонами.

Значения параметров по умолчанию в ECMAScript 6

Спецификация ECMAScript 6 упрощает объявление значений параметров по умолчанию, добавляя оператор инициализации, который выполняется, если параметр не был явно передан в вызов. Например:

```
function makeRequest(url, timeout = 2000, callback = function() {}) {

    // остальная часть функции
}
```

Эта функция ожидает получить только первый обязательный параметр, а два других имеют значения по умолчанию. Новый синтаксис помогает существенно сократить тело функции за счет отсутствия необходимости добавлять дополнительный код проверки отсутствующих значений.

Когда `makeRequest()` вызывается со всеми тремя параметрами, значения по умолчанию просто не используются. Например:

```
// timeout и callback получают значения по умолчанию
makeRequest("/foo");

// callback получит значение по умолчанию
makeRequest("/foo", 500);

// значения по умолчанию не используются
makeRequest("/foo", 500, function(body) {
    doSomething(body);
});
```

Параметр `url` в ECMAScript 6 считается обязательным. Именно поэтому во все три вызова `makeRequest()` в примере выше передается строка `"/foo"`. Два параметра со значениями по умолчанию рассматриваются как необязательные.

Значения по умолчанию можно указать для любых аргументов, в том числе и для тех, которые предшествуют аргументам, не имеющим значений по умолчанию в определении функции. Например, следующее определение вполне допустимо:

```
function makeRequest(url, timeout = 2000, callback) {  
    // оставшая часть функции  
}
```

В данном случае значение по умолчанию для параметра `timeout` будет использоваться, только если второй аргумент не будет передан в вызов функции или в нем явно будет передано значение `undefined`, например:

```
// timeout получит значение по умолчанию  
makeRequest("/foo", undefined, function(body) {  
    doSomething(body);  
});  
  
// timeout получит значение по умолчанию  
makeRequest("/foo");  
  
// значение по умолчанию для timeout не используется  
makeRequest("/foo", null, function(body) {  
    doSomething(body);  
});
```

Здесь механизм значений параметров по умолчанию считает значение `null` допустимым, вследствие чего в третьем вызове `makeRequest()` значение по умолчанию для параметра `timeout` использоваться не будет.

Как значения параметров по умолчанию влияют на объект `arguments`

Имейте в виду, что при наличии значений параметров по умолчанию объект `arguments` действует иначе. В ECMAScript 5 в нестрогом режиме объект `arguments` отражает изменения в именованных параметрах функции. Следующий код иллюстрирует это поведение:

```
function mixArgs(first, second) {  
    console.log(first === arguments[0]);  
    console.log(second === arguments[1]);  
    first = "c";  
    second = "d";  
    console.log(first === arguments[0]);  
    console.log(second === arguments[1]);  
}  
  
mixArgs("a", "b");
```

Этот пример выведет следующее:

```
true  
true  
true  
true
```

В нестрогом режиме выполнения объект `arguments` всегда отражает изменения в именованных параметрах. Таким образом, когда параметрам `first` и `second` присваиваются новые значения, `arguments[0]` и `arguments[1]` также изменяются, благодаря чему все операторы сравнения `===` возвращают `true`.

Однако строгий режим в ECMAScript 5 ликвидирует эту путаницу. В строгом режиме объект `arguments` не отражает изменений в именованных параметрах. Ниже вновь приводится определение функции `mixArgs()`, но на этот раз с включением строгого режима:


```
function mixArgs(first, second) {
    "use strict";

    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}
```

```
mixArgs("a", "b");
```

Вызов этой версии `mixArgs()` выведет следующее:

```
true
true
false
false
```

На этот раз изменения в параметрах `first` и `second` не повлияли на объект `arguments`. В результате мы получили более ожидаемый вывод.

Объект `arguments` в функции, использующей значения параметров по умолчанию ECMAScript 6, всегда действует, как в строгом режиме ECMAScript 5, даже если он не был включен явно. Присутствие значений параметров по умолчанию отключает синхронизацию объекта `arguments` с изменениями в именованных параметрах. Это тонкая, но важная деталь, влияющая на порядок использования объекта `arguments`. Взгляните на следующий фрагмент:

```
// нестрогий режим
function mixArgs(first, second = "b") {
    console.log(arguments.length);
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
    first = "c";
    second = "d"
    console.log(first === arguments[0]);
    console.log(second === arguments[1]);
}
```

```
mixArgs("a");
```

Он выведет следующее:

```
1
true
false
false
false
```

В этом примере свойство `arguments.length` имеет значение 1, потому что в вызов `mixArgs()` передан только один аргумент. Это также означает, что `arguments[1]` имеет значение `undefined`, что вполне ожидаемо в случае, когда в функцию передается только один аргумент. Значит, значение `first` равно значению `arguments[0]`. Изменение параметров `first` и `second` не оказывает влияния на `arguments`. Такое поведение проявляется в обоих режимах — строгом и нестрогом, — поэтому можно смело положиться на тот факт, что `arguments` всегда отражает начальное состояние аргументов.

Выражения в параметрах по умолчанию

Самое интересное, что значения параметров по умолчанию не обязательно должны быть элементарными значениями. Значение по умолчанию можно получить, например, вызовом функции, как в следующем примере:

```
function getValue() {
    return 5;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));    // 2
console.log(add(1));       // 6
```

Здесь, если опустить последний аргумент, для получения значения по умолчанию будет вызвана функция `getValue()`. Имейте в виду, что `getValue()` вызывается только в момент вызова `add()` без второго параметра, а не когда она объявляется и анализируется интерпретатором. Это означает, что если переписать функцию `getValue()` иначе, можно организовать получение разных значений по умолчанию. Например:

```
let value = 5;

function getValue() {
    return value++;
}

function add(first, second = getValue()) {
    return first + second;
}

console.log(add(1, 1));    // 2
console.log(add(1));       // 6
console.log(add(1));       // 7
```

В данном примере переменная `value` получает начальное значение 5 и увеличивается на единицу с каждым вызовом `getValue()`. Первый вызов `add(1)` возвращает 6, а второй вызов `add(1)` возвращает 7, потому что значение переменной `value` увеличилось. Так как значение по умолчанию для параметра `second` вычисляется в момент вызова функции, мы можем изменить его в любой момент.

ВНИМАНИЕ

Будьте внимательны, используя функции для получения значений по умолчанию. Если вы забудете поставить круглые скобки после имени функции в определении параметра, например `second = getValue`, в параметр будет записана ссылка на функцию, а не результат ее вызова.

Эта особенность поведения значений параметров по умолчанию открывает еще одну полезную возможность. Значение предыдущего параметра можно использовать как значение по умолчанию любого из последующих параметров. Например:

```
function add(first, second = first) {  
    return first + second;  
}
```

```
console.log(add(1, 1));    // 2  
console.log(add(1));       // 2
```

В этом примере параметр `second` имеет значение по умолчанию, равное значению параметра `first`, вследствие чего, если передать только один аргумент, второй получит точно такое же значение. Поэтому оба вызова — `add(1, 1)` и `add(1)` — возвращают одно и то же значение 2. Этот прием можно развить дальше и передать `first` в вызов функции, возвращающей значение по умолчанию для `second`, например:

```
function getValue(value) {  
    return value + 5;  
}  
  
function add(first, second = getValue(first)) {  
    return first + second;  
}
```

```
console.log(add(1, 1));    // 2  
console.log(add(1));       // 7
```

В этом примере параметр `second` получает значение, возвращаемое вызовом `getValue(first)`, поэтому, хотя вызов `add(1, 1)` по-прежнему возвращает 2, вызов `add(1)` возвращает уже 7(1+6).

Возможность ссылки на параметры в значениях по умолчанию поддерживается только для предыдущих аргументов, поэтому в предыдущем параметре нельзя сослаться на последующий. Например:

```
function add(first = second, second) {  
    return first + second;  
}  
  
console.log(add(1, 1));           // 2  
console.log(add(undefined, 1));   // вызовет ошибку
```

Вызов `add(undefined, 1)` завершается ошибкой, потому что параметр `second` определяется после `first` и не может служить для него значением по умолчанию. Чтобы понять, почему так происходит, вспомните временные мертвые зоны.

Временная мертвая зона для параметров по умолчанию

В главе 1 при обсуждении операторов `let` и `const` мы познакомились с понятием временной мертвой зоны (TDZ). Значения по умолчанию для параметров также имеют временную мертвую зону, находясь в которой они остаются недоступными.

По аналогии с объявлением `let` каждый параметр создает новую привязку к идентификатору, на которую нельзя сослаться до инициализации, не вызвав ошибку. Инициализация параметра происходит в момент вызова функции — либо явно переданным значением, либо значением по умолчанию.

С целью исследования временной мертвой зоны значений параметров по умолчанию вернемся к примеру из предыдущего раздела:

```
function getValue(value) {  
    return value + 5;
```

```
}

function add(first, second = getValue(first)) {
    return first + second;
}

console.log(add(1, 1));    // 2
console.log(add(1));       // 7
```

Вызовы `add(1, 1)` и `add(1)` фактически выполняют следующий код, чтобы создать значения для параметров `first` и `second`:

```
// Представление вызова add(1, 1) в коде на JavaScript
let first = 1;
let second = 1;

// Представление вызова add(1) в коде на JavaScript
let first = 1;
let second = getValue(first);
```

При первом вызове функции `add()` во временную мертвую зону для параметров добавляются привязки `first` и `second` (подобно тому, как действует `let`). Параметр `second` может инициализироваться значением `first`, потому что к этому моменту `first` уже инициализирован, но обратный порядок инициализации невозможен. Теперь рассмотрим переписанную версию функции `add()`:

```
function add(first = second, second) {
    return first + second;
}

console.log(add(1, 1));           // 2
console.log(add(undefined, 1));   // вызовет ошибку
```

В этом примере вызовы `add(1, 1)` и `add(undefined, 1)` отображаются в следующий код (неявно):

```
// Представление вызова add(1, 1) в коде на JavaScript
let first = 1;
let second = 1;

// Представление вызова add(undefined, 1) в коде на JavaScript
let first = second;
let second = 1;
```

Вызов `add(undefined, 1)` в этом примере приводит к ошибке, потому что параметр `second` еще не инициализирован к моменту, когда инициализируется параметр `first`. В этот момент `second` находится во временной мертвой зоне, и любые ссылки на него вызывают ошибку. Этот пример отражает поведение привязок `let`, которое обсуждалось в главе 1.

ПРИМЕЧАНИЕ

Параметры функции имеют собственную область видимости и собственную временную мертвую зону отдельно от области видимости тела функции. Это означает, что в значении параметра по умолчанию нельзя обращаться ни к каким переменным, объявленным в теле функции.

Неименованные параметры

До сих пор в этой главе рассматривались только примеры параметров с именами, присвоенными в определении функции. Однако в JavaScript количество параметров, которые можно передавать функциям, не ограничивается количеством объявленных именованных параметров. Вы всегда можете передать меньше или больше параметров, чем формально определено. Значения параметров по умолчанию ясно показывают, что функция способна принимать меньшее количество параметров, но ECMAScript 6 помогает не менее ясно показать, что функция может также принимать больше параметров, чем определено.

Неименованные параметры в ECMAScript 5

Раньше для доступа ко всем параметрам, переданным в вызов функции, в JavaScript поддерживался объект `arguments`. Несмотря на то что в большинстве случаев этот объект прекрасно справлялся со своей задачей, для работы с ним требовалось писать громоздкий код. Например, взгляните на следующий пример, извлекающий значения из объекта `arguments`:

```
function pick(object) {
    let result = Object.create(null);

    // обход параметров, начиная со второго
    for (let i = 1, len = arguments.length; i < len; i++) {
        result[arguments[i]] = object[arguments[i]];
    }

    return result;
}

let book = {
    title: "Understanding ECMAScript 6",
    author: "Nicholas C. Zakas",
    year: 2016
};

let bookData = pick(book, "author", "year");

console.log(bookData.author);    // "Nicholas C. Zakas"
console.log(bookData.year);      // 2016
```

Эта функция имитирует метод `pick()` из библиотеки *Underscore.js*, который возвращает копию заданного объекта с определенным множеством свойств из оригинального объекта. В этом примере функция определяет только один аргумент, который, как ожидается, является объектом, чьи свойства требуется скопировать. Все остальные аргументы представляют имена свойств для копирования в результат.

Обратите внимание на пару особенностей функции `pick()`. Во-первых, при беглом осмотре трудно понять, что функция способна обрабатывать более одного параметра. Да, можно определить дополнительные параметры, но таким способом нельзя ясно указать, что функция может принимать произвольное количество параметров. Во-вторых, поскольку первый параметр является именованным и используется непосредственно, чтобы выявить имена свойств для копирования, вы вынуждены извлекать значения из объекта `arguments`, начиная с индекса 1, а не 0. Запомнить индекс, с которого следует начинать, несложно, но это еще одна деталь, о которой нужно помнить.

Для решения этой проблемы в ECMAScript 6 были добавлены остаточные параметры.

Остаточные параметры

Остаточный параметр (*rest parameter*) отмечается троеточием (...), предшествующим именованному параметру. Такой именованный параметр превращается в массив `Array`, содержащий все остальные параметры, переданные в вызов функции, откуда и взялось название *остаточные*. Например, с использованием остаточных параметров функцию `pick()` можно переписать, как показано ниже:

```
function pick(object, ...keys) {
  let result = Object.create(null);

  for (let i = 0, len = keys.length; i < len; i++) {
    result[keys[i]] = object[keys[i]];
  }
  return result;
}
```

В этой версии функции `pick()` `keys` является остаточным параметром, содержащим все остальные параметры, переданные в вызов после `object` (в отличие от объекта `arguments`, который содержит все параметры, включая первый). Это означает, что можно организовать обход элементов массива `keys` от начала до конца, ни о чем не беспокоясь. Как дополнительное преимущество объявление такой функции ясно говорит, что она может обрабатывать произвольное количество параметров.

ПРИМЕЧАНИЕ

Остаточные параметры не влияют на свойство `length` функции, которое определяет количество именованных параметров. В этом примере свойство `length` функции `pick()` имеет значение 1, потому что учитывает только присутствие параметра `object`.

Ограничения остаточных параметров

Остаточные параметры имеют два ограничения. Первое ограничение: в функции может иметься только один остаточный параметр, и он должен объявляться последним. Например, следующий код не будет работать:

```
// Синтаксическая ошибка: именованные параметры не могут следовать
// за остаточными параметрами
function pick(object, ...keys, last) {
  let result = Object.create(null);

  for (let i = 0, len = keys.length; i < len; i++) {
    result[keys[i]] = object[keys[i]];
  }

  return result;
}
```

Здесь параметр `last` следует за остаточным параметром `keys`, что вызывает синтаксическую ошибку.

Второе ограничение: остаточные параметры не могут использоваться в методах записи свойств в литералах объектов. Следовательно, следующий код также вызовет синтаксическую ошибку:

```
let object = {  
  
    // Синтаксическая ошибка: нельзя использовать остаточные параметры  
    // в методах записи свойств в литералах объектов  
    set name(...value) {  
        // тело метода  
    }  
};
```

Это ограничение объясняется тем, что в литералах объектов методы записи свойств могут принимать только один аргумент. Остаточные параметры по определению представляют бесконечное множество аргументов, поэтому их нельзя использовать в данном контексте.

Как остаточные параметры влияют на объект `arguments`

Остаточные параметры в JavaScript предназначались для использования вместо объекта `arguments`. Первоначально спецификация ECMAScript 4 ликвидировала объект `arguments` и добавила остаточные параметры, чтобы дать возможность передавать в функции неограниченное количество аргументов. Но ECMAScript 4 так и не была стандартизована, хотя сама идея сохранилась и была воплощена в ECMAScript 6, несмотря на то что объект `arguments` не был убран из языка.

Объект `arguments` действует параллельно остаточным параметрам и хранит все аргументы, переданные в вызов функции, как показано в следующем примере.

```
function checkArgs(...args) {  
    console.log(args.length);  
    console.log(arguments.length);  
    console.log(args[0], arguments[0]);  
    console.log(args[1], arguments[1]);  
}  
  
checkArgs("a", "b");
```

Данный вызов функции `checkArgs()` выведет следующее:

```
2  
2  
a a  
b b
```

Объект `arguments` всегда правильно отражает параметры, переданные в функцию, независимо от наличия остаточного параметра.

Дополнительные возможности конструктора Function

Конструктор `Function` — редко используемая часть JavaScript, позволяющая динамически создавать новые функции. В аргументах конструктору передаются параметры и тело функции, все в виде строк. Например:

```
var add = new Function("first", "second", "return first + second");  
  
console.log(add(1, 1));    // 2
```

ECMAScript 6 расширяет поддержку аргументов конструктора `Function`, разрешая определять значения параметров по умолчанию и остаточные параметры. Чтобы определить значение по умолчанию, достаточно добавить знак «равно» после имени параметра и значение, как показано ниже:

```
var add = new Function("first", "second = first",  
    "return first + second");
```

```
console.log(add(1, 1));    // 2  
console.log(add(1));       // 2
```

В этом примере параметру `second` присваивается значение параметра `first`, если функция вызывается с единственным параметром. Синтаксис тот же самый, как в обычном объявлении функции без использования конструктора `Function`.

Чтобы объявить остаточные параметры, просто добавьте `...` перед последним параметром, как в следующем примере:

```
var pickFirst = new Function("...args", "return args[0]");  
  
console.log(pickFirst(1, 2));    // 1
```

Этот код создаст функцию с единственным остаточным параметром, которая возвращает первый переданный ей аргумент. Помимо значений по умолчанию и остаточных параметров конструктор `Function` поддерживает все те возможности, которые имеет обычная декларативная форма определения функций.

Оператор расширения

С остаточными параметрами тесно связан оператор расширения (`spread`). Но, в отличие от остаточных параметров, позволяющих объединить в массив множество независимых аргументов, оператор расширения дает возможность передать массив, который должен быть разбит и передан в функцию в виде отдельных аргументов. Рассмотрим встроенный метод `Math.max()`, принимающий произвольное количество аргументов и возвращающий наибольшее значение. Ниже показан простейший случай применения этого метода:

```
let value1 = 25,  
    value2 = 50;  
  
console.log(Math.max(value1, value2));    // 50
```

Когда имеется всего два значения, как в данном примере, пользоваться методом `Math.max()` очень просто. Достаточно передать два значения и получить наибольшее значение. Но представьте, что имеется целый массив значений и требуется получить наибольшее из них. Метод `Math.max()` не принимает массивы, поэтому в ECMAScript 5 и более ранних версиях приходилось выполнять поиск в массиве вручную или применять метод `apply()`, как показано ниже:

```
let values = [25, 50, 75, 100]  
  
console.log(Math.max.apply(Math, values));    // 100
```

Это вполне работоспособное решение, но применение метода `apply()` таким способом несколько сбивает с толку. Фактически этот прием затрудняет выяснение истинного назначения кода.

Оператор расширения, появившийся в ECMAScript 6, существенно упрощает подобные ситуации. Вместо вызова `apply()` можно просто передать массив в вызов `Math.max()`, добавив перед ним то же троеточие `...`, как при объявлении остаточных параметров. Движок JavaScript разобьет такой массив на отдельные аргументы и передаст их в функцию, как показано ниже:


```
let values = [25, 50, 75, 100]

// эквивалентно вызову
// console.log(Math.max(25, 50, 75, 100));
console.log(Math.max(...values));    // 100
```

Теперь вызов `Math.max()` выглядит более понятным и не требует в простых операциях прибегать к таким сложностям, как передача привязки `this` (первый аргумент `Math.max.apply()` в предыдущем примере).

Оператор расширения можно также использовать с другими аргументами. Предположим, необходимо, чтобы наименьшим числом, возвращаемым методом `Math.max()`, был 0 (чтобы исключить возврат отрицательных чисел). В этом случае данное значение можно передать в отдельном аргументе и использовать оператор расширения для остальных, как показано ниже:

```
let values = [-25, -50, -75, -100]

console.log(Math.max(...values, 0));    // 0
```

В этом примере в последнем аргументе методу `Math.max()` передается число 0, при этом все предыдущие аргументы передаются с помощью оператора расширения.

Применение оператора расширения для передачи аргументов упрощает применение массивов в качестве аргументов функций. Вы сами убедитесь, что во многих случаях это отличная замена методу `apply()`.

Свойство name

Идентификация функций в JavaScript может вызывать определенные сложности, учитывая богатство способов определения функций. Кроме того, широкое применение анонимных функций-выражений усложняет отладку, потому что приходится тратить много времени на чтение и расшифровку трассировки стека. По этим причинам спецификация ECMAScript 6 добавила во все функции свойство `name`.

Выбор соответствующих имен

Все функции в программах на ECMAScript 6 имеют соответствующее значение в свойстве `name`. Чтобы понять суть, разберем следующий пример, где объявляются функция и функция-выражение и выводятся значения их свойств `name`:

```
function doSomething() {
    // пустая
}

var doAnotherThing = function() {
    // пустая
};

console.log(doSomething.name);    // "doSomething"
console.log(doAnotherThing.name);    // "doAnotherThing"
```

В этом примере свойство `name` функции `doSomething()` получило значение `"doSomething"`, потому что оно указано в определении функции. Свойство `name` анонимной функции-выражения `doAnotherThing()` получило значение `"doAnotherThing"`, потому что это имя имеет переменная, которой была присвоена ссылка на функцию.

Специальные случаи свойства `name`

Достаточно просто определить имена функций и функций-выражений из их объявлений, но ECMAScript 6 пошла дальше и гарантирует выбор соответствующих имен для всех функций. Для иллюстрации рассмотрим следующую программу:

```
var doSomething = function doSomethingElse() {
    // пустая
};

var person = {
    get firstName() {
        return "Nicholas"
    },
    sayName: function() {
        console.log(this.name);
    }
}

console.log(doSomething.name);           // "doSomethingElse"
console.log(person.sayName.name);       // "sayName"
console.log(person.firstName.name);     // "get firstName"
```

В этом примере свойство `doSomething.name` получило значение `"doSomethingElse"`, потому что функция-выражение имеет имя и это имя имеет более высокий приоритет, чем имя переменной, которой присваивается ссылка на функцию. Свойство `name` функции `person.sayName()` получило значение `"sayName"`, потому что оно было получено интерпретатором из литерала объекта. Функция `person.firstName` в действительности является методом чтения свойства, поэтому она получила имя `"get firstName"`, отмечающее эту особенность. Аналогично методы записи предваряются префиксом `"set"`.

Существует еще пара специальных случаев выбора имен для функций. Функции, созданные с помощью `bind()`, получают имена с префиксом `"bound"` и пробелом после него, а функции, созданные с помощью конструктора `Function`, получают имя `"anonymous"`, как в следующем примере:

```
var doSomething = function() {
    // пустая
};

console.log(doSomething.bind().name);    // "bound doSomething"
console.log((new Function()).name);     // "anonymous"
```

Свойство `name` связанной функции всегда получает значение, состоящее из значения свойства `name` оригинальной функции и префикса `"bound "`, поэтому связанная версия функции `doSomething()` получит имя `"bound doSomething"`.

Имейте в виду, что значение свойства `name` любой функции необязательно ссылается на переменную с тем же именем. Свойство `name` добавлено исключительно для предоставления дополнительной информации при отладке, поэтому нет никакой возможности использовать значение `name`, чтобы получить ссылку на функцию.

Двойственная природа функций

В ECMAScript 5 и более ранних версиях функции имели двойственную природу — они могли вызываться с ключевым словом `new` или без него. При вызове с ключевым словом `new` ссылка

`this` внутри функции указывала на новый объект, и этот новый объект возвращался функцией, как иллюстрирует следующий пример:

```
function Person(name) {
    this.name = name;
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas");

console.log(person);           // "[Object object]"
console.log(notAPerson);       // "undefined"
```

Для создания `notAPerson` функция `Person()` вызывается без ключевого слова `new` и возвращает значение `undefined` (а в нестрогом режиме дополнительно устанавливает значение свойства `name` глобального объекта). Первая заглавная буква в имени `Person` — единственный признак, широко используемый в программах на JavaScript и указывающий на то, что функция предполагает вызов с ключевым словом `new`. Эта путаница, связанная с двойственной природой функций, стала причиной некоторых изменений в ECMAScript 6.

Функции в JavaScript имеют два разных внутренних метода: `[[Call]]` и `[[Construct]]`. Когда функция вызывается без ключевого слова `new`, выполняется метод `[[Call]]`, который выполняет тело функции, присутствующее в коде. Когда функция вызывается с ключевым словом `new`, выполняется метод `[[Construct]]`. Метод `[[Construct]]` создает новый объект, который еще называют экземпляром, и затем выполняет тело функции, предварительно присвоив этот экземпляр ссылке `this`. Функции, имеющие метод `[[Construct]]`, называют *конструкторами*.

Имейте в виду, что не все функции имеют метод `[[Construct]]` и, соответственно, не все функции могут вызываться с ключевым словом `new`. Например, стрелочные функции, которые обсуждаются в разделе «Стрелочные функции» ниже, не имеют метода `[[Construct]]`.

Как в ECMAScript 5 определить, каким способом вызвана функция

Чтобы определить, была ли вызвана функция с ключевым словом `new` (то есть как конструктор) или без него, в ECMAScript 5 обычно используется оператор `instanceof`, например:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;           // с ключевым словом new
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person("Nicholas"); // вызовет ошибку
```

Здесь выясняется, указывает ли ссылка `this` на экземпляр конструктора, и если указывает, то выполнение продолжается как обычно. В противном случае, если `this` не является экземпляром `Person`, возбуждается ошибка. Эта проверка опирается на тот факт, что метод `[[Construct]]` создает новый экземпляр `Person` и присваивает его ссылке `this`. К сожалению, данное решение не является абсолютно надежным, потому что `this` может ссылаться на экземпляр `Person` и без использования ключевого слова `new`, как в следующем примере:

```
function Person(name) {
    if (this instanceof Person) {
        this.name = name;
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // работает!
```

Вызов `Person.call()` получает переменную `person` в первом аргументе, то есть внутри функции `Person` ссылка `this` будет указывать на `person`. Функция не в состоянии отличить вызов `Person.call()` (или `Person.apply()`) с экземпляром `Person` от вызова с применением ключевого слова `new`.

Метасвойство `new.target`

Чтобы дать функциям возможность определить факт вызова с ключевым словом `new`, ECMAScript 6 определяет новое метасвойство `new.target`. *Метасвойство* — это необъектное свойство с дополнительной информацией о его цели (такой, как `new`). Когда вызывается метод `[[Construct]]` функции, в `new.target` сохраняется цель оператора `new`. Такой целью обычно является конструктор вновь созданного экземпляра объекта, который будет присвоен ссылке `this` в теле функции. Когда вызывается метод `[[Call]]`, `new.target` получает значение `undefined`.

Это новое метасвойство позволяет с полной уверенностью отличить вызов функции с ключевым словом `new`, как показано ниже:

```
function Person(name) {
    if (typeof new.target !== "undefined") {
        this.name = name;
    } else {
        throw new Error("You must use new with Person.")
    }
}

var person = new Person("Nicholas");
var notAPerson = Person.call(person, "Michael");    // ошибка!
```

Благодаря использованию `new.target` вместо `this instanceof Person` конструктор `Person` теперь будет вызывать ошибку при любом вызове без ключевого слова `new`.

С помощью `new.target` можно также проверить вызов конкретного конструктора. Например, взгляните на следующий пример:

```
function Person(name) {
    if (typeof new.target === Person) {
        this.name = name;
    } else {
        throw new Error("You must use new with Person.")
    }
}

function AnotherPerson(name) {
    Person.call(this, name);
}
```

```
}

var person = new Person("Nicholas");
var anotherPerson = new AnotherPerson("Nicholas");    // ошибка!
```

Для нормальной работы этот код требует, чтобы метасвойство `new.target` имело тип `Person`. Вызов `Person.call(this, name)`, который производится в вызове `new AnotherPerson("Nicholas")`, возбudit ошибку, потому что в этом случае `new.target` внутри конструктора `Person` получит значение `undefined` (так как он был вызван без ключевого слова `new`).

ВНИМАНИЕ

Использование `new.target` за пределами функции вызовет синтаксическую ошибку.

Добавив метасвойство `new.target`, спецификация ECMAScript 6 помогла разрешить некоторую неоднозначность с вызовами функций. Следуя в этом же направлении, ECMAScript 6 разрешила еще одну неоднозначность, имевшую место в языке: объявление функций внутри блоков.

Функции уровня блоков

В ECMAScript 3 и более ранних версиях попытка объявить функцию внутри блока (*функцию уровня блока*) теоретически должна вызывать синтаксическую ошибку, но все браузеры поддерживали такую возможность. К сожалению, все браузеры, допускавшие подобный синтаксис, вели себя немного по-разному, поэтому считалось, что лучше избегать объявления функций в блоках (а при необходимости использовать функции-выражения).

В попытке устранить несовместимость в поведении спецификация ECMAScript 5 потребовала в строгом режиме возбуждать ошибку всякий раз, когда функция объявляется внутри блока, например:

```
"use strict";

if (true) {

    // вызовет синтаксическую ошибку в ES5, но не в ES6
    function doSomething() {
        // пустая
    }
}
```

В ECMAScript 5 этот код вызывает синтаксическую ошибку. В ECMAScript 6 функция `doSomething()` интерпретируется как объявление на уровне блока и может вызываться в том же блоке, где она объявлена. Например:

```
"use strict";

if (true) {
    console.log(typeof doSomething);    // "function"

    function doSomething() {
        // пустая
    }
    doSomething();
}

console.log(typeof doSomething);        // "undefined"
```

Объявления функций на уровне блока поднимаются в начало этого блока, поэтому оператор `typeof doSomething` вернул `"function"`, хотя он и находится выше объявления функции. Как только выполнение блока `if` завершится, функция `doSomething()` станет недоступна.

Когда использовать функции уровня блока

Функции уровня блока напоминают функции-выражения в инструкции `let` тем, что определение функции удаляется, когда поток выполнения покидает блок, в котором она определена. Основное отличие состоит в том, что определения функций уровня блока поднимаются в начало вмещающего блока. Функции-выражения, объявленные с помощью `let`, не поднимаются, как иллюстрирует следующий пример:

```
"use strict";

if (true) {
  console.log(typeof doSomething);    // вызовет ошибку

  let doSomething = function () {
    // пустая
  }
  doSomething();
}

console.log(typeof doSomething);
```

Этот код прекратит выполнение, когда встретится оператор `typeof doSomething`, потому что инструкция `let` к этому моменту еще не была выполнена и идентификатор `doSomething()` находится во временной мертвой зоне. Зная это отличие, вы сможете выбирать между функциями уровня блока и выражениями `let` в зависимости от того, желаете ли получить эффект подъема.

Функции уровня блока в нестрогом режиме

Спецификация ECMAScript 6 разрешила также объявлять функции уровня блока в нестрогом режиме, но их поведение в этом случае немного отличается. Вместо подъема в начало вмещающего блока объявления поднимаются в начало вмещающей функции или глобальной области видимости. Например:

```
// Поведение в ECMAScript 6
if (true) {

  console.log(typeof doSomething);    // "function"

  function doSomething() {
    // пустая
  }

  doSomething();
}

console.log(typeof doSomething);      // "function"
```

В этом примере `doSomething()` поднимается в начало глобальной области видимости, поэтому оказывается доступной за пределами блока `if`. Спецификация ECMAScript 6 стандартизовала

это поведение, чтобы ликвидировать несовместимость в поведении браузеров, так что все среды выполнения, совместимые с ECMAScript 6, должны действовать одинаково.

Поддержка функций уровня блока расширяет возможности объявления функций в JavaScript, но в ECMAScript 6 появились и совершенно новые способы объявления функций.

Стрелочные функции

Одним из интереснейших новшеств в ECMAScript 6 являются *стрелочные функции* (*arrow functions*). Стрелочные функции, как следует из названия, объявляются с применением нового синтаксиса, в котором используется оператор стрелка (`=>`). Кроме того, стрелочные функции имеют несколько важных отличий от традиционных функций JavaScript:

- **Отсутствие привязок `this`, `super`, `arguments` и `new.target`.** Значения `this`, `super`, `arguments` и `new.target` внутри стрелочных функций определяются ближайшей областью видимости традиционной (не являющейся стрелочной) функции JavaScript. (Ссылка `super` обсуждается в главе 4.)
- **Не могут вызываться с ключевым словом `new`.** Стрелочные функции не имеют метода `[[Construct]]` и поэтому не могут использоваться как конструкторы. Попытка вызвать стрелочную функцию с ключевым словом `new` приводит к ошибке.
- **Отсутствует прототип.** Поскольку стрелочные функции не могут вызываться с ключевым словом `new`, они не нуждаются в прототипах. Стрелочные функции не имеют свойства `prototype`.
- **Нельзя изменить `this`.** Значение `this` внутри стрелочных функций нельзя изменить. Оно остается неизменным в течение всего жизненного цикла функции.
- **Отсутствует объект `arguments`.** Так как стрелочные функции не имеют привязки `arguments`, они могут использовать только именованные и остаточные параметры для доступа к аргументам функции.
- **Не поддерживают повторяющиеся имена параметров.** Стрелочные функции не могут иметь именованные параметры с повторяющимися именами в обоих режимах, строгом и нестрогом, тогда как нестрелочные функции не могут иметь именованные параметры с повторяющимися именами только в строгом режиме.

Эти отличия имеют несколько объяснений. Во-первых, привязка `this` считается основным источником ошибок в JavaScript. Внутри функции очень легко потерять контроль над значением `this`, что может привести к непредсказуемому поведению программы, и стрелочные функции устраняют эту проблему. Во-вторых, благодаря ограничению стрелочных функций простым выполнением кода с неизменным значением `this`, движки JavaScript легко могут оптимизировать операции с ним, в отличие от обычных функций, которые могут использоваться как конструкторы или изменять `this` иными способами.

Остальные отличия также обусловлены стремлением уменьшить вероятность появления ошибок и избавиться от неоднозначностей в стрелочных функциях. Благодаря этому движки JavaScript имеют более широкое поле для оптимизации выполнения стрелочных функций.

ПРИМЕЧАНИЕ

Стрелочные функции имеют свойство `name`, которое подчиняется тем же правилам, что и свойство `name` других функций.

Синтаксис стрелочных функций

Синтаксис стрелочных функций имеет много разновидностей в зависимости от целей, которых вы пытаетесь достичь. Объявление всех разновидностей начинается со списка параметров функции, за которым следует стрелка и тело функции. Список параметров и тело функции могут принимать разные формы в зависимости от назначения. Например, следующая стрелочная функция имеет один параметр и просто возвращает его:

```
let reflect = value => value;

// фактически эквивалентна следующей функции:
let reflect = function(value) {
  return value;
};
```

Если стрелочная функция имеет единственный параметр, он может использоваться непосредственно, без дополнительных синтаксических конструкций. Проще говоря, стрелка вычисляет выражение справа и возвращает результат. Хотя в этой стрелочной функции отсутствует инструкция `return`, она вернет значение своего первого аргумента.

Если функция имеет несколько параметров, их нужно заключить в круглые скобки, как в следующем примере:

```
let sum = (num1, num2) => num1 + num2;

// фактически эквивалентна следующей функции:
let sum = function(num1, num2) {
  return num1 + num2;
};
```

Функция `sum()` просто находит сумму двух аргументов и возвращает результат. Единственное отличие стрелочной функции `sum()` от предыдущей функции `reflect()` состоит в том, что параметры заключены в круглые скобки и перечисляются через запятую (как в объявлениях традиционных функций).

Если функция не имеет параметров, в объявление необходимо добавлять пустые круглые скобки, как показано ниже:

```
let getName = () => "Nicholas";

// фактически эквивалентна следующей функции:
let getName = function() {
  return "Nicholas";
};
```

Когда требуется определить тело функции, больше напоминающее тело традиционной функции и, возможно, включающее несколько выражений, его необходимо заключить в фигурные скобки и явно определить возвращаемое значение, как в следующей версии `sum()`:

```
let sum = (num1, num2) => {
  return num1 + num2;
};

// фактически эквивалентна следующей функции:
let sum = function(num1, num2) {
  return num1 + num2;
};
```


Внутри фигурных скобок допустимо практически все то же самое, что и в традиционных функциях, за исключением объекта `arguments`, который отсутствует в стрелочных функциях.

Если потребуется объявить функцию, которая ничего не делает, достаточно просто указать пустые фигурные скобки, например:

```
let doNothing = () => ;

// фактически эквивалентна следующей функции:
let doNothing = function() ;
```

Во всех случаях, встречавшихся до сих пор, фигурные скобки обозначали тело функции. Но если стрелочная функция должна вернуть литерал объекта, этот литерал должен заключаться в круглые скобки. Например:

```
let getItem = id => ({ id: id, name: "Temp" });

// фактически эквивалентна следующей функции:
let getItem = function(id) {
  return {
    id: id,
    name: "Temp"
  };
};
```

Заключение литерала объекта в круглые скобки сообщает интерпретатору, что фигурные скобки заключают литерал объекта, а не тело функции.

Создание выражений немедленно вызываемых функций

В JavaScript широко используются выражения немедленно вызываемых функций (Immediately Invoked Function Expressions, IIFE). Это позволяет объявить анонимную функцию и сразу же вызвать ее, минуя этап сохранения ссылки. Данный прием особенно удобен, когда требуется создать область видимости, закрытую от остальной программы. Например:

```
let person = function(name) {

  return {
    getName: function() {
      return name;
    }
  };
}("Nicholas");

console.log(person.getName()); // "Nicholas"
```

В этом примере выражение немедленно вызываемой функции создает объект с методом `getName()`. Этот метод использует аргумент `name` как возвращаемое значение, фактически создавая приватное свойство `name` в возвращаемом объекте.

Тот же результат можно получить с помощью стрелочной функции, заключив ее в круглые скобки:

```
let person = ((name) => {  
    return {  
        getName: function() {  
            return name;  
        }  
    };  
})( "Nicholas");  
  
console.log(person.getName());    // "Nicholas"
```

Обратите внимание, что в круглые скобки заключено только определение стрелочной функции, но не ее аргумент ("Nicholas"). Этим стрелочные функции отличаются от обычных функций, для которых в круглые скобки может заключаться только объявление функции или объявление с аргументами.

Отсутствие привязки **this**

Привязка **this** внутри функций считается одним из основных источников ошибок в JavaScript. Поскольку значение **this** внутри одной и той же функции может изменяться в зависимости от контекста вызова, функция может по ошибке изменить совсем не тот объект, который предполагался. Взгляните на следующий пример:

```
let PageHandler = {  
    id: "123456",  
    init: function() {  
        document.addEventListener("click", function(event) {  
            this.doSomething(event.type);    // ошибка  
        }, false);  
    };  
    doSomething: function(type) {  
        console.log("Handling " + type + " for " + this.id);  
    }  
};
```

В этом фрагменте определяется объект **PageHandler**, предназначенный для обслуживания взаимодействий со страницей. Метод **init()** устанавливает обработчик, который вызывает метод **this.doSomething()**. Но этот код будет работать не так, как предполагается.

Вызов **this.doSomething()** приведет к ошибке, потому что **this** ссылается на объект, представляющий собой цель события (в данном случае **document**), а не на объект **PageHandler**. При выполнении этого кода возбуждается ошибка, когда будет вызван обработчик события, потому что целевой объект **document** не имеет метода **this.doSomething()**.

Эту проблему можно решить, явно связав значение **this** со ссылкой на **PageHandler** с помощью метода **bind()** функции, как показано ниже:

```
let PageHandler = {  
    id: "123456",
```

```
init: function() {
    document.addEventListener("click", (function(event) {
        this.doSomething(event.type);    // ошибки нет
    }).bind(this), false);
},

doSomething: function(type) {
    console.log("Handling " + type + " for " + this.id);
}
};
```

Теперь код будет работать как ожидается, хотя выглядит немного странно. Вызывая `bind(this)`, вы фактически создаете новую функцию, в которой ссылка `this` связана с текущим значением `this`, указывающим на `PageHandler`. Чтобы избежать создания дополнительной функции, этот код лучше исправить, применив стрелочную функцию.

Стрелочные функции не имеют привязки `this`, поэтому значение `this` внутри стрелочной функции определяется только цепочкой областей видимости. Если стрелочная функция находится внутри нестрелочной функции, ссылка `this` в ней будет иметь то же значение, что и во вмещающей функции; в противном случае ссылка `this` будет не определена. Ниже демонстрируется один из вариантов реализации желаемого поведения с применением стрелочной функции:

```
let PageHandler = {

    id: "123456",

    init: function() {
        document.addEventListener("click",
            event => this.doSomething(event.type), false);
    },

    doSomething: function(type) {
        console.log("Handling " + type + " for " + this.id);
    }
};
```

В этом примере обработчик события, вызывающий `this.doSomething()`, является стрелочной функцией. Ссылка `this` в нем имеет то же значение, что и в методе `init()`, поэтому данная версия действует точно так же, как версия, использующая `bind(this)`. Хотя метод `doSomething()` ничего не возвращает, его вызов является единственной инструкцией в теле функции, поэтому нет необходимости заключать его в фигурные скобки.

Стрелочные функции — это «разовые» функции, поэтому они не могут определять новые типы; это очевидно вытекает из отсутствия свойства `prototype`, которым обладают обычные функции. Попытка использовать ключевое слово `new` со стрелочной функцией, как в следующем примере, вызовет ошибку:

```
var MyType = () => {},
    object = new MyType();    // ошибка - стрелочные функции нельзя вызывать
                               // с ключевым словом 'new'
```

В этом примере вызов `new MyType()` приведет к ошибке, потому что `MyType` — стрелочная функция, не имеющая метода `[[Construct]]`. Знание того, что стрелочные функции не могут использоваться с ключевым словом `new`, позволяет движкам JavaScript оптимизировать их выполнение.

Кроме того, поскольку значение `this` определяется функцией, вмещающей стрелочную функцию, внутри стрелочной функции нельзя изменить это значение вызовом `call()`, `apply()` или `bind()`.

Стрелочные функции и массивы

Лаконичный синтаксис стрелочных функций делает их идеальным инструментом для обработки массивов. Например, когда требуется отсортировать массив с применением нестандартной функции сравнения, обычно пишется примерно такой код:

```
var result = values.sort(function(a, b) {  
    return a - b;  
});
```

Довольно много кода для простой процедуры. Сравните это с более краткой версией на основе стрелочной функции:

```
var result = values.sort((a, b) => a - b);
```

Методы массивов, принимающие функции обратного вызова, такие как `sort()`, `map()` и `reduce()`, могут получать дополнительные выгоды от лаконичного синтаксиса стрелочных функций, с помощью которого внешне сложные процедуры можно выразить в простом коде.

Отсутствие привязки arguments

Несмотря на то что стрелочные функции не имеют собственного объекта `arguments`, они могут использовать объект `arguments` вмещающей функции. Этот объект останется доступным стрелочной функции, где бы она ни вызывалась потом. Например:

```
function createArrowFunctionReturningFirstArg() {  
    return () => arguments[0];  
}  
  
var arrowFunction = createArrowFunctionReturningFirstArg(5);  
  
console.log(arrowFunction());    // 5
```

Стрелочная функция, созданная внутри `createArrowFunctionReturningFirstArg()`, ссылается на элемент `arguments[0]`. Эта ссылка хранит первый аргумент, переданный в вызов `createArrowFunctionReturningFirstArg()`. Если позднее вызвать эту стрелочную функцию, она вернет число 5 — значение первого аргумента, переданного в вызов `createArrowFunctionReturningFirstArg()`. Хотя стрелочная функция больше не находится в области видимости функции, создавшей ее, объект `arguments` все еще остается для нее доступным.

Идентификация стрелочных функций

Несмотря на отличающийся синтаксис, стрелочные функции остаются функциями и идентифицируются как таковые. Взгляните на следующий код:

```
var comparator = (a, b) => a - b;  
  
console.log(typeof comparator);    // "function"  
console.log(comparator instanceof Function);    // true
```

Как показывает вывод `console.log()`, оба оператора, `typeof` и `instanceof`, воспринимают стрелочные функции как любые другие функции.

Кроме того, как и любые другие функции, стрелочные функции имеют методы `call()`, `apply()` и `bind()`, правда, при этом они не затрагивают привязку `this`. Вот несколько примеров:

```
var sum = (num1, num2) => num1 + num2;

console.log(sum.call(null, 1, 2));      // 3
console.log(sum.apply(null, [1, 2]));   // 3

var boundSum = sum.bind(null, 1, 2);

console.log(boundSum());                // 3
```

Функцию `sum()` можно вызвать с использованием `call()` и `apply()` для передачи аргументов, как любую другую функцию. Метод `bind()` создает `boundSum()` с двумя связанными аргументами 1 и 2, поэтому их не требуется передавать непосредственно.

Стрелочные функции можно применять везде, где используются анонимные функции-выражения, например в качестве функций обратного вызова. Следующий раздел охватывает еще одно важное новшество, появившееся в ECMAScript 6, но оно носит внутренний характер и не имеет соответствующей синтаксической конструкции.

Оптимизация хвостовых вызовов

Самым интересным, пожалуй, изменением в ECMAScript 6, коснувшимся функций, является оптимизация хвостовых вызовов в недрах движка. *Хвостовым (tail)* называется вызов функции, являющийся последней инструкцией в другой функции, как в следующем примере:

```
function doSomething() {
    return doSomethingElse();    // хвостовой вызов
}
```

В движках, совместимых с ECMAScript 5, хвостовые вызовы функций обрабатываются точно так же, как любые другие: создается новый кадр стека и помещается в стек вызовов для представления вызова функции. Это означает, что предыдущий кадр продолжает храниться в памяти, что может превратиться в проблему, когда стек вызовов разрастается до слишком больших размеров.

Отличия хвостовых вызовов в ECMAScript 6

ECMAScript 6 предусматривает уменьшение размера стека вызовов для определенных хвостовых вызовов в строгом режиме (в нестрогом режиме оптимизация хвостовых вызовов не выполняется). При такой оптимизации вместо создания нового кадра стека для хвостового вызова очищается и повторно используется текущий кадр стека, если выполняются следующие условия:

- Хвостовой вызов не обращается к переменным в текущем кадре стека (то есть вызываемая функция не образует замыкания).
- Функция, производящая хвостовой вызов, не предусматривает никаких других операций после возврата из хвостового вызова.
- Результат хвостового вызова возвращается как результат вызывающей функции.

Например, следующий фрагмент соответствует всем трем критериям и может быть оптимизирован:

```
"use strict";

function doSomething() {
    // оптимизируется
    return doSomethingElse();
}
```

Эта функция производит хвостовой вызов `doSomethingElse()`, сразу же возвращает результат и не имеет переменных в локальной области видимости, к которым могла бы обратиться функция `doSomethingElse()`. Но стоит внести одно маленькое изменение — не вернуть результат, и оптимизация выполнена не будет:

```
"use strict";

function doSomething() {
    // не оптимизируется - отсутствует оператор return
    doSomethingElse();
}
```

Если функция выполняет какие-то операции после возврата из хвостового вызова, такой вызов не оптимизируется:

```
"use strict";

function doSomething() {
    // не оптимизируется - после вызова выполняется операция сложения
    return 1 + doSomethingElse();
}
```

В этом примере перед возвратом результата к результату вызова `doSomethingElse()` прибавляется число 1, и этого достаточно, чтобы оптимизация не была выполнена.

Другая типичная причина отключения оптимизации по неосторожности — сохранение результата вызова функции в переменной с последующим ее возвратом, например:

```
"use strict";

function doSomething() {
    // не оптимизируется - вызов выполняется не последним
    var result = doSomethingElse();
    return result;
}
```

Этот вызов нельзя оптимизировать, потому что результат вызова `doSomethingElse()` не возвращается немедленно.

Труднее всего, пожалуй, избежать образования замыканий. Так как функция, образующая замыкание, обращается к переменным во внешней области видимости, оптимизация хвостового вызова может быть отключена. Например:

```
"use strict";

function doSomething() {
    var num = 1,
        func = () => num;
```

```
// не оптимизируется – функция образует замыкание
return func();
}
```

В этом примере функция `func()` обращается к локальной переменной `num` и тем самым образует замыкание. Хотя результат вызова `func()` немедленно возвращается вмещающей функцией, оптимизация не выполняется из-за наличия ссылки на переменную `num`.

Как использовать оптимизацию хвостовых вызовов

Оптимизация хвостовых вызовов выполняется в недрах движка, поэтому нет смысла думать о ней, если только вы не пытаетесь оптимизировать функцию. Основными кандидатами на оптимизацию хвостовых вызовов являются рекурсивные функции, потому что именно в этом случае оптимизация дает наибольший эффект. Рассмотрим функцию, вычисляющую факториалы:

```
function factorial(n) {

    if (n <= 1) {
        return 1;
    } else {
        // не оптимизируется – после вызова выполняется операция умножения
        return n * factorial(n - 1);
    }
}
```

Эта версия функции не может быть оптимизирована, потому что после рекурсивного вызова `factorial()` требуется выполнить умножение. Если `n` — очень большое число, количество кадров в стеке вызовов может увеличиться до такой степени, что вызовет его переполнение.

Чтобы оптимизация стала возможной, необходимо гарантировать отсутствие операции умножения после последнего вызова функции. Для этого можно использовать параметр по умолчанию и вынести операцию умножения за пределы инструкции `return`. Получившаяся функция переносит временный результат в следующую итерацию. Новая версия действует точно так же, как предыдущая, но может быть оптимизирована движком ECMAScript 6:

```
function factorial(n, p = 1) {

    if (n <= 1) {
        return 1 * p;
    } else {
        let result = n * p;

        // оптимизируется
        return factorial(n - 1, result);
    }
}
```

В новой версии `factorial()` появился второй параметр `p` со значением по умолчанию 1. Он хранит результат предыдущего умножения, чтобы с его помощью можно было вычислить следующий результат без другого вызова функции. Когда `n` больше 1, сначала выполняется умножение, а затем результат передается в вызов `factorial()` во втором аргументе. Такая реорганизация позволяет движку ECMAScript 6 оптимизировать рекурсивный вызов.

Вспоминайте об оптимизации хвостовых вызовов всякий раз, когда будете писать очередную рекурсивную функцию, потому что она поможет вам получить значительный прирост производительности, особенно когда применяется к функциям, выполняющим массивные вычисления.

ВНИМАНИЕ

На момент написания этих строк оптимизация хвостовых вызовов в ECMAScript 6 подвергалась переработке. Вполне возможно, что для большей ясности включение этой оптимизации будет реализовано с применением какого-то особого синтаксиса. Продолжающиеся обсуждения могут вылиться в изменения в ECMAScript 8 (ECMAScript 2017).

В заключение

Функции не подверглись существенным изменениям в спецификации ECMAScript 6, но в ней появились дополнительные изменения, упрощающие работу с функциями.

Значения параметров по умолчанию позволяют указать, какое значение должен получить конкретный аргумент, если он не был передан в вызов. До ECMAScript 6 для этого требовалось писать дополнительный код внутри функции, чтобы проверить наличие аргументов и присвоить им соответствующие значения по умолчанию.

Остаточные параметры дают возможность определить массив, куда должны помещаться все остальные аргументы. Возможность использовать настоящий массив и явно указать, какие параметры должны в него помещаться, делает остаточные параметры намного более гибким решением, чем объект `arguments`.

Оператор расширения дополняет механизм остаточных параметров и позволяет разложить массив на отдельные параметры при вызове функции. До ECMAScript 6 имелось только два способа передать элементы массива в виде отдельных параметров: вручную определив каждый параметр или с помощью `apply()`. С помощью оператора расширения можно легко передать массив в любую функцию, не заботясь о привязке `this` в этой функции.

Дополнительное свойство `name` создавалось с целью упростить идентификацию функций для нужд отладки и интерпретации. Также в ECMAScript 6 официально утверждено поведение функций уровня блока, чтобы исключить появление синтаксических ошибок в строгом режиме.

В ECMAScript 6 поведение функции определяется методом `[[Call]]`, когда функция вызывается как обычно, и методом `[[Construct]]`, когда она вызывается с ключевым словом `new`. Метасвойство `new.target` позволяет определить внутри функции, как она была вызвана — с ключевым словом `new` или без него.

Самое значительное изменение в поддержке функций в ECMAScript 6 — это появление стрелочных функций. Основная цель стрелочных функций — заменить анонимные функции-выражения. Стрелочные функции имеют более лаконичный синтаксис, лексическую привязку `this` и не имеют объекта `arguments`. Кроме того, стрелочные функции не могут изменять привязку `this` и потому не способны выступать в роли конструкторов.

Оптимизация хвостовых вызовов позволяет изменять вызовы некоторых функций с целью уменьшить размер стека вызовов и объем используемой памяти, а также предотвратить переполнение стека. Эта оптимизация применяется движком автоматически, когда это возможно; однако вы можете решить переписать рекурсивные функции, чтобы получить преимущества этой оптимизации.

Расширенные возможности объектов

В ECMAScript 6 большое внимание уделяется улучшению объектов, и это вполне объяснимо, потому что практически любое значение в JavaScript представлено объектом того или иного типа. Количество объектов, используемых разработчиками в программах на JavaScript, увеличивается вместе со сложностью приложений. Учитывая большое количество объектов в программах, становится очевидной необходимость увеличения их эффективности.

ECMAScript 6 внесла много разных усовершенствований в объекты — от простых синтаксических расширений до новых возможностей управления и взаимодействий с ними, — и в данной главе мы подробно рассмотрим эти усовершенствования.

Категории объектов

В JavaScript используется разная терминология для описания объектов, определяемых стандартом и добавляемых средами выполнения, такими как браузеры. Спецификация ECMAScript 6 четко определяет каждую категорию объектов. Поэтому знание данной терминологии совершенно необходимо, чтобы освоить язык в целом. Определены следующие категории объектов:

- **Обычные объекты.** Обладают всеми чертами поведения объектов, которые поддерживаются в JavaScript.
- **Необычные объекты.** Обладают чертами поведения, отличающими их от обычных объектов.
- **Стандартные объекты.** Определяемые спецификацией ECMAScript 6, такие как `Array`, `Date` и т. д. Стандартные объекты могут быть обычными и необычными.
- **Встроенные объекты.** Определяются средой выполнения JavaScript, в которой действует сценарий. Все стандартные объекты одновременно являются встроенными.

Я буду пользоваться этими терминами на протяжении всей книги для описания различных объектов, которые определяет ECMAScript 6.

Расширение синтаксиса литералов объектов

Литералы объектов — один из самых распространенных шаблонов программирования на JavaScript. На этом синтаксисе основан формат JSON, и он используется чуть ли не в каждом файле с кодом на JavaScript в Интернете. Популярность литералов объектов объясняется лаконичностью синтаксиса

описания объектов, для создания которых в иных условиях может потребоваться написать несколько строк кода. К счастью для разработчиков, ECMAScript 6 дополнила синтаксис литералов объектов новыми и еще более лаконичными расширениями.

Сокращенный синтаксис инициализации свойств

В ECMAScript 5 и в более ранних версиях литералы объектов были простыми коллекциями пар имя/значение, вследствие чего код инициализации свойств мог дублироваться. Например:

```
function createPerson(name, age) {  
    return {  
        name: name,  
        age: age  
    };  
}
```

Функция `createPerson()` создает объект, имена свойств которого совпадают с именами параметров функции. В результате код содержит повторяющиеся идентификаторы `name` и `age`, из которых один принадлежит свойству объекта, а другой определяет значение для этого свойства. Ключ `name` в возвращаемом объекте получает значение, хранящееся в переменной `name`, а ключ `age` — в переменной `age`.

В ECMAScript 6 можно убрать такие повторения, когда имена свойств совпадают с именами локальных переменных, применив сокращенный синтаксис *инициализации свойств*. Когда имя свойства совпадает с именем локальной переменной, в объявлении объекта можно оставить одно имя свойства без двоеточия и значения для инициализации. Например, функцию `createPerson()` можно переписать для ECMAScript 6, как показано ниже:

```
function createPerson(name, age) {  
    return {  
        name,  
        age  
    };  
}
```

Если свойство в литерале объекта описывается только именем, движок JavaScript проверит присутствие переменной с тем же именем в окружающей области видимости. И если такая переменная имеется, ее значение будет присвоено одноименному свойству литерала объекта. В данном примере свойство `name` литерала объекта получит значение локальной переменной `name`.

Сокращенный синтаксис определения свойств делает инициализацию литералов объектов еще более лаконичной и помогает уменьшить вероятность опечаток в именах. Присваивание свойствам значений одноименных локальных переменных — весьма распространенный прием в JavaScript, поэтому данное расширение является очень полезным дополнением.

Сокращенный синтаксис определения методов

В ECMAScript 6 также был усовершенствован синтаксис определения методов в литералах объектов. В ECMAScript 5 и предшествующих версиях, чтобы добавить метод в объект, требуется указать его имя и затем полное определение функции, как показано ниже:

```
var person = {  
    name: "Nicholas",  
    sayName: function() {  
        console.log(this.name);  
    }  
};
```

В ECMAScript 6 добавлен более краткий синтаксис, исключаящий двоеточие и ключевое слово `function`. Поэтому предыдущий пример можно переписать так:

```
var person = {
  name: "Nicholas",
  sayName() {
    console.log(this.name);
  }
};
```

Этот синтаксис, который также называют *сокращенным синтаксисом определения методов*, создает в объекте `person` тот же самый метод, что и в предыдущем примере. Свойству `sayName()` присваивается анонимная функция-выражение, и оно получает все те же характеристики, что и функция `sayName()` в ECMAScript 5. Единственное отличие состоит в том, что методы, объявленные с помощью сокращенного синтаксиса, могут использовать `super` (обсуждается ниже в разделе «Простой доступ к прототипу с помощью ссылки `super`»), тогда как методы, объявленные обычным способом, такой возможности не имеют.

ПРИМЕЧАНИЕ

Свойство `name` метода, созданного с помощью сокращенного синтаксиса, получает строку с именем, указанным перед круглыми скобками. В данном примере свойство `name` метода `person.sayName()` получит значение `"sayName"`.

Вычисляемые имена свойств

ECMAScript 5 и более ранние версии поддерживали использование вычисляемых имен свойств экземпляров объектов, если они указывались в квадратных скобках вместо применения точечной нотации. В квадратных скобках имена свойств можно определять с помощью переменных и литералов строк, которые могут содержать символы, вызывающие синтаксическую ошибку при их присутствии в идентификаторах. Например:

```
var person = {},
    lastName = "last name";

person["first name"] = "Nicholas";
person[lastName] = "Zakas";

console.log(person["first name"]); // "Nicholas"
console.log(person[lastName]);    // "Zakas"
```

Так как переменной `lastName` присвоена строка `"last name"`, оба свойства в этом примере получают имена, содержащие пробел, что делает невозможным обращение к ним с использованием точечной нотации. Квадратные скобки, напротив, позволяют использовать любые символы в именах свойств, поэтому присваивание свойству `"first name"` значения `"Nicholas"` и свойству `"last name"` значения `"Zakas"` выполняется без ошибок.

Кроме того, в литералах объектов допускается использовать строки в качестве имен свойств:

```
var person = {
  "first name": "Nicholas"
};

console.log(person["first name"]); // "Nicholas"
```

Этот прием можно использовать, когда имена свойств известны заранее и могут быть представлены в виде строковых литералов. Однако если имя свойства `"first name"` хранится в переменной (как в предыдущем примере) или должно вычисляться, вы не сможете определить его в литерале объекта в ECMAScript 5.

В ECMAScript 6 вычисляемые имена свойств стали частью синтаксиса литералов объектов и используют ту же форму записи с квадратными скобками, которая применяется для обращения к свойствам экземпляров объектов по вычисляемым именам. Например:

```
let lastName = "last name";

let person = {
  "first name": "Nicholas",
  [lastName]: "Zakas"
};

console.log(person["first name"]); // "Nicholas"
console.log(person[lastName]);    // "Zakas"
```

Квадратные скобки внутри литерала объекта подсказывают, что имя свойства вычисляется и интерпретируется как строка. Это означает, что в квадратных скобках можно использовать даже выражения, как в следующем примере:

```
var suffix = " name";

var person = {
  ["first" + suffix]: "Nicholas",
  ["last" + suffix]: "Zakas"
};

console.log(person["first name"]); // "Nicholas"
console.log(person["last name"]);  // "Zakas"
```

Имена этих свойств вычисляются как `"first name"` и `"last name"`, и эти строки можно использовать для обращения к свойствам позднее. Любые выражения, допустимые в квадратных скобках при обращении к свойствам экземпляров объектов, допустимы также внутри литералов объектов.

Новые методы

Разработчики ECMAScript, начиная с ECMAScript 5, стремились избежать добавления новых глобальных функций и методов в `Object.prototype`. Вместо этого, когда требовалось добавить в стандарт новые методы, они добавляли их в соответствующие существующие объекты. В результате глобальный объект `Object` получал новые методы, только когда они были неуместны в других объектах. ECMAScript 6 ввела в глобальный объект `Object` два новых метода, которые упрощают решение некоторых задач.

Метод `Object.is()`

Когда в JavaScript требуется сравнить два значения, обычно используют оператор равенства (`==`) или идентичности (`===`). Многие разработчики предпочитают последний, чтобы исключить приведение типов при сравнении. Но даже оператор идентичности не всегда дает точный результат.

Например, значения `+0` и `-0` оператор `===` считает идентичными, хотя движок JavaScript представляет их по-разному. Кроме того, выражение `NaN === NaN` вернет `false`, и по этой причине для правильной проверки значения `NaN` требуется использовать метод `isNaN()`.

Спецификация ECMAScript 6 добавила метод `Object.is()`, чтобы компенсировать неточность оператора идентичности в других случаях. Этот метод принимает два аргумента и возвращает `true`, если значения эквивалентны. Два значения считаются эквивалентными, если они имеют один и тот же тип и одну и ту же величину. Например:

```
console.log(+0 == -0);           // true
console.log(+0 === -0);          // true
console.log(Object.is(+0, -0));  // false

console.log(NaN == NaN);         // false
console.log(NaN === NaN);        // false
console.log(Object.is(NaN, NaN)); // true

console.log(5 == 5);             // true
console.log(5 == "5");           // true
console.log(5 === 5);            // true
console.log(5 === "5");          // false
console.log(Object.is(5, 5));     // true
console.log(Object.is(5, "5"));   // false
```

Во многих случаях `Object.is()` действует в точности как оператор `===`. Единственное отличие: значения `+0` и `-0` он считает разными, а два значения `NaN` — одинаковыми. Но появление этого метода не означает, что операторы равенства стали не нужны. Используйте метод `Object.is()` вместо `==` или `===`, только если корректность работы вашего кода зависит от этих особых случаев.

Метод `Object.assign()`

Примеси (mixins) — один из самых популярных шаблонов компоновки объектов в JavaScript. В примеси один объект получает свойства и методы другого объекта. Многие библиотеки на JavaScript включают метод `mixin`, напоминающий следующий:

```
function mixin(receiver, supplier) {
    Object.keys(supplier).forEach(function(key) {
        receiver[key] = supplier[key];
    });

    return receiver;
}
```

Функция `mixin()` выполняет обход собственных свойств объекта `supplier` и копирует их в объект `receiver` (при этом выполняется поверхностное копирование, когда копируются только ссылки, если значением свойства является ссылка на объект). Это позволяет придать объекту `receiver` новые свойства без наследования, как в следующем примере:

```
function EventTarget() { /*...*/ }
EventTarget.prototype = {
    constructor: EventTarget,
    emit: function() { /*...*/ },
    on: function() { /*...*/ }
};
```

```
var myObject = {};  
mixin(myObject, EventTarget.prototype);  
  
myObject.emit("somethingChanged");
```

Здесь `myObject` получает черты поведения от объекта `EventTarget.prototype`, дающие ему возможность публиковать события и подписываться на них, используя методы `emit()` и `on()` соответственно.

Шаблон «Примесь» приобрел настолько большую популярность, что в ECMAScript 6 был добавлен метод `Object.assign()`, который действует точно так же, принимая объект-приемник и произвольное количество объектов-поставщиков и затем возвращая объект-приемник. Смена имени с `mixin()` на `assign()` отражает фактически выполняемую операцию. Так как функция `mixin()` использует оператор присваивания (`=`), она не может копировать методы доступа к свойствам в принимающий объект. Именно это обстоятельство повлияло на выбор имени `Object.assign()`.

ПРИМЕЧАНИЕ

Похожие методы с той же функциональностью в разных библиотеках могут иметь другие имена; популярными альтернативами являются методы `extend()` и `mix()`. Вслед за методом `Object.assign()` вскоре в ECMAScript 6 был добавлен метод `Object.mixin()`. Основное отличие между ними заключается в том, что `Object.mixin()` копирует также методы доступа к свойствам, но потом этот метод был удален из-за проблем, возникающих при использовании `super` (обсуждается ниже в разделе «Простой доступ к прототипу с помощью ссылки `super`»).

Метод `Object.assign()` можно без ограничений использовать взамен функции `mixin()`. Например:

```
function EventTarget() { /*...*/ }  
EventTarget.prototype = {  
  constructor: EventTarget,  
  emit: function() { /*...*/ },  
  on: function() { /*...*/ }  
}  
  
var myObject = {}  
Object.assign(myObject, EventTarget.prototype);  
  
myObject.emit("somethingChanged");
```

Метод `Object.assign()` принимает произвольное количество объектов-поставщиков и присваивает объекту-приемнику их свойства в том порядке, в каком объекты-поставщики перечислены в вызове метода. Это означает, что свойство второго поставщика может затереть в объекте-приемнике свойство первого поставщика, как в следующем примере:

```
var receiver = {};  
  
Object.assign(receiver,  
  {  
    type: "js",  
    name: "file.js"  
  },  
  {
```

```
        type: "css"
    }
};

console.log(receiver.type);    // "css"
console.log(receiver.name);    // "file.js"
```

Свойство `receiver.type` получило значение `"css"`, потому что свойство второго объекта-поставщика затерло ранее скопированное значение из первого объекта-поставщика.

Метод `Object.assign()` не является существенным расширением ECMAScript 6, но формализует распространенную функциональность, которую можно найти во многих библиотеках на JavaScript.

РАБОТА СО СВОЙСТВАМИ, ИМЕЮЩИМИ МЕТОДЫ ДОСТУПА

Имейте в виду, что метод `Object.assign()` не создает в объекте-приемнике методы доступа к свойствам, имеющиеся в объекте-поставщике. Копирование выполняется с помощью оператора присваивания, поэтому свойства с методами доступа объекта-поставщика превращаются в обычные свойства-данные объекта-приемника. Например:

```
var receiver = {},
    supplier = {
        get name() {
            return "file.js"
        }
    };

Object.assign(receiver, supplier);

var descriptor = Object.getOwnPropertyDescriptor(receiver, "name");

console.log(descriptor.value);    // "file.js"
console.log(descriptor.get);      // undefined
```

В этом примере объект `supplier` имеет свойство с именем `name` с методом доступа. После вызова метода `Object.assign()` объект `receiver` получит обычное свойство `name` со значением `"file.js"`, потому что в вызове `Object.assign()` обращение к `supplier.name` вернет `"file.js"`.

Дубликаты свойств в литералах объектов

В строгом режиме в ECMAScript 5 выполняется проверка наличия дубликатов свойств в литералах объектов и возбуждение ошибки, если дубликаты имеются. Например, следующий фрагмент вызывал ошибку:

```
"use strict";

var person = {
    name: "Nicholas",
    name: "Greg"    // синтаксическая ошибка в строгом режиме ES5
};
```

При попытке выполнить его в строгом режиме в ECMAScript 5 присутствие второго свойства `name` вызовет синтаксическую ошибку. Но в ECMAScript 6 проверка дубликатов свойств была убрана. Теперь ни в одном из режимов проверка дубликатов свойств не выполняется. Вместо этого фактическим значением свойства становится последнее указанное значение, как показано ниже:

```
"use strict";

var person = {
  name: "Nicholas",
  name: "Greg"    // нет ошибки в строгом режиме ES6
};

console.log(person.name);    // "Greg"
```

В этом примере свойство `person.name` получит значение `"Greg"`, потому что оно было присвоено последним.

Порядок перечисления собственных свойств

ECMAScript 5 не определяет порядок перечисления свойств объекта, хотя создатели движков JavaScript перечисляют их в определенном порядке. Однако ECMAScript 6 строго определила порядок, в котором должны возвращаться собственные свойства при перечислении. Это повлияло на результат, возвращаемый методами `Object.getOwnPropertyNames()` и `Reflect.ownKeys` (описываются в главе 12), а также на порядок обработки свойств в методе `Object.assign()`.

В общем случае собственные свойства перечисляются в следующем порядке:

1. Все числовые ключи в порядке возрастания.
2. Все строковые ключи в порядке добавления в объект.
3. Все символические ключи (рассматриваются в главе 6) в порядке добавления в объект.

Например:

```
var obj = {
  a: 1,
  0: 1,
  c: 1,
  2: 1,
  b: 1,
  1: 1
};

obj.d = 1;

console.log(Object.getOwnPropertyNames(obj).join(""));    // "012acbd"
```

Вызов `Object.getOwnPropertyNames()` вернул свойства объекта `obj` в порядке: 0, 1, 2, a, c, b, d. Обратите внимание, что числовые ключи группируются вместе и сортируются, несмотря на то что в литерале объекта они располагаются в ином порядке. За числовыми ключами следуют строковые ключи в порядке их добавления в объект `obj`. Ключи, объявленные в литерале объекта, перечисляются первыми, за ними следуют динамические ключи, добавленные позднее (в данном

случае d).

ПРИМЕЧАНИЕ

Цикл `for-in` по-прежнему не гарантирует какой-то определенный порядок перечисления свойств, потому что он по-разному реализован в разных движках JavaScript. Методы `Object.keys()` и `JSON.stringify()` перечисляют свойства в том же (неопределенном) порядке, что и цикл `for-in`.

Стандартизация порядка перечисления свойств кажется не особенно заметным изменением в работе JavaScript, тем не менее нередко можно встретить программы, полагающиеся на вполне определенный порядок. Определив порядок перечисления, ECMAScript 6 гарантирует, что код на JavaScript, зависящий от порядка перечисления свойств, будет работать правильно в любом окружении.

Расширения в прототипах

Прототипы — основа наследования в JavaScript, и спецификация ECMAScript 6 продолжила вносить усовершенствования в прототипы. Ранние версии JavaScript строго ограничивали круг допустимых операций с прототипами. Однако по мере развития языка и освоения программистами особенностей работы с прототипами стало очевидно, что разработчикам требуется более полный контроль над прототипами и простые средства для работы с ними. В результате в ECMAScript 6 появилось несколько усовершенствований прототипов.

Смена прототипа объекта

Обычно объект получает прототип в момент создания посредством конструктора или метода `Object.create()`. Идея, состоящая в том, что прототип объекта остается неизменным после создания экземпляра, была одним из основополагающих допущений в программировании на JavaScript вплоть до ECMAScript 5. Спецификация ECMAScript 5 добавила метод `Object.getPrototypeOf()` для получения прототипа любого заданного объекта, но в ней все еще отсутствовал стандартный способ смены прототипа объекта после создания экземпляра.

Спецификация ECMAScript 6 изменила это допущение, добавив метод `Object.setPrototypeOf()`, позволяющий заменять прототип заданного объекта. Метод `Object.setPrototypeOf()` принимает два аргумента: объект, прототип которого требуется сменить, и объект, который должен стать прототипом объекта в первом аргументе. Например:

```
let person = {
  getGreeting() {
    return "Hello";
  }
};

let dog = {
  getGreeting() {
    return "Woof";
  }
};

// прототип - person
let friend = Object.create(person);
console.log(friend.getGreeting()); // "Hello"
```

```
console.log(Object.getPrototypeOf(friend) === person);    // true

// сменить прототип на dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());                        // "Woof"
console.log(Object.getPrototypeOf(friend) === dog);        // true
```

В этом фрагменте определяются два простых объекта: `person` и `dog`. Оба объекта имеют метод `getGreeting()`, возвращающий строку. Объект `friend` первоначально наследует объект `person`, соответственно его метод `getGreeting()` выводит "Hello". После смены прототипа на объект `dog` вызов `person.getGreeting()` вывел "Woof", потому что первоначальная связь с объектом `person` была разорвана.

Фактическая ссылка на прототип объекта хранится во внутреннем свойстве `[[Prototype]]`. Метод `Object.getPrototypeOf()` возвращает значение свойства `[[Prototype]]` и метод `Object.setPrototypeOf()` изменяет значение свойства `[[Prototype]]`. Однако эти два метода — не единственный способ доступа к значению `[[Prototype]]`.

Простой доступ к прототипу с помощью ссылки `super`

Как отмечалось выше, прототипы играют очень важную роль в JavaScript, и в ECMAScript 6 была проделана большая работа с целью еще больше упростить их использование. Еще одно усовершенствование заключается в добавлении ссылки `super`, упрощающей доступ к возможностям прототипа объекта. Например, чтобы переопределить метод экземпляра объекта и обеспечить в нем вызов одноименного метода прототипа, в ECMAScript 5 требуется выполнить следующие операции:

```
let person = {
  getGreeting() {
    return "Hello";
  }
};

let dog = {
  getGreeting() {
    return "Woof";
  }
};

let friend = {
  getGreeting() {
    return Object.getPrototypeOf(this).getGreeting.call(this) + ", hi!";
  }
};

// сменить прототип на person
Object.setPrototypeOf(friend, person);
console.log(friend.getGreeting());                // "Hello, hi!"
console.log(Object.getPrototypeOf(friend) === person); // true

// сменить прототип на dog
Object.setPrototypeOf(friend, dog);
console.log(friend.getGreeting());                // "Woof, hi!"
console.log(Object.getPrototypeOf(friend) === dog);  // true
```

В этом примере метод `getGreeting()` объекта `friend` вызывает одноименный метод прототипа. Метод `Object.getPrototypeOf()` гарантирует вызов метода прототипа, а затем в вывод добавляется дополнительная строка. Добавка `.call(this)` гарантирует, что ссылка `this` внутри метода прототипа будет иметь правильное значение.

Необходимость помнить о применении метода `Object.getPrototypeOf()` и добавлении `.call(this)` к вызову метода прототипа немного усложняет дело, поэтому в ECMAScript 6 была введена ссылка `super`. Выражаясь простым языком, `super` — это указатель на текущий прототип объекта, фактически это значение `Object.getPrototypeOf(this)`. Зная это, можно упростить реализацию метода `getGreeting()`:

```
let friend = {
  getGreeting() {
    // в предыдущем примере ту же операцию выполняла инструкция:
    // Object.getPrototypeOf(this).getGreeting.call(this)
    return super.getGreeting() + ", hi!";
  }
};
```

Вызов `super.getGreeting()` в данном контексте действует в точности как `Object.getPrototypeOf(this).getGreeting.call(this)`. Аналогично с помощью ссылки `super` можно вызвать любой метод прототипа объекта при условии, что вызов производится в методе, объявленном с применением сокращенного синтаксиса. Попытка использовать `super` в методе, объявленном с помощью традиционного синтаксиса, вызовет синтаксическую ошибку, как показано в следующем примере:

```
let friend = {
  getGreeting: function() {
    // синтаксическая ошибка
    return super.getGreeting() + ", hi!";
  }
};
```

В этом примере используется именованное свойство с функцией, и вызов `super.getGreeting()` вызывает синтаксическую ошибку, потому что использование ссылки `super` недопустимо в этом контексте.

Ссылка `super` может очень пригодиться, когда имеется несколько уровней наследования, потому что в этом случае `Object.getPrototypeOf()` дает верный результат не во всех случаях. Например:

```
let person = {
  getGreeting() {
    return "Hello";
  }
};

// прототип - person
let friend = {
  getGreeting() {
    return Object.getPrototypeOf(this).getGreeting.call(this) + ", hi!";
  }
};

Object.setPrototypeOf(friend, person);
```

```
// прототип - friend
let relative = Object.create(friend);

console.log(person.getGreeting());    // "Hello"
console.log(friend.getGreeting());    // "Hello, hi!"
console.log(relative.getGreeting());  // ошибка!
```

Когда вызывается метод `relative.getGreeting()`, обращение к `Object.getPrototypeOf()` приводит к ошибке. Причина в том, что `this` ссылается на `relative`, а прототипом `relative` является объект `friend`. Когда происходит вызов `friend.getGreeting().call()` со ссылкой на `relative` в `this`, процесс заикливается и рекурсивный вызов продолжается до тех пор, пока не возникнет ошибка переполнения стека.

Эту проблему сложно решить в ECMAScript 5, но в ECMAScript 6 с помощью ссылки `super` она решается просто:

```
let person = {
  getGreeting() {
    return "Hello";
  }
};

// прототип - person
let friend = {
  getGreeting() {
    return super.getGreeting() + ", hi!";
  }
};
Object.setPrototypeOf(friend, person);

// прототип - friend
let relative = Object.create(friend);

console.log(person.getGreeting());    // "Hello"
console.log(friend.getGreeting());    // "Hello, hi!"
console.log(relative.getGreeting());  // "Hello, hi!"
```

Поскольку `super` не является динамической ссылкой, она всегда указывает на правильный объект. В данном случае `super.getGreeting()` всегда ссылается на `person.getGreeting()` независимо от того, как много объектов унаследовало данный метод.

Формальное определение метода

До появления ECMAScript 6 понятие «метод» не имело формального определения. Методы были всего лишь свойствами объектов, хранящими ссылки на функции вместо данных. Спецификация ECMAScript 6 формально определяет метод как функцию, имеющую внутреннее свойство `[[HomeObject]]`, ссылающееся на объект, которому принадлежит метод. Взгляните на следующий фрагмент:

```
let person = {

  // метод
  getGreeting() {
```

```
        return "Hello";
    }
};

//не метод
function shareGreeting() {
    return "Hi!";
}
```

В этом примере определяется объект `person` с единственным методом `getGreeting()`. Свойство `[[HomeObject]]` метода `getGreeting()` получит ссылку на `person` в силу прямой принадлежности объекту. Но функция `shareGreeting()` не получит свойства `[[HomeObject]]`, потому что объявлена за пределами какого-либо объекта. В большинстве случаев это различие не играет большой роли, но становится очень важным, когда используется ссылка `super`.

Любое обращение к `super` связано с обращением к свойству `[[HomeObject]]` с целью определить, что делать. Первый шаг в этом процессе — вызов `Object.getPrototypeOf()` для `[[HomeObject]]`, чтобы получить ссылку на прототип. Далее выполняется поиск в прототипе функции с тем же именем. Затем устанавливается привязка `this` и вызывается метод. Взгляните на следующий пример:

```
let person = {
    getGreeting() {
        return "Hello";
    }
};

// прототип - person
let friend = {
    getGreeting() {
        return super.getGreeting() + ", hi!";
    }
};
Object.setPrototypeOf(friend, person);

console.log(friend.getGreeting());    // "Hello, hi!"
```

Вызов `friend.getGreeting()` вернул строку, состоящую из значения, возвращаемого методом `person.getGreeting()`, и строки `hi!`. Свойство `[[HomeObject]]` метода `friend.getGreeting()` ссылается на объект `friend`, а прототипом объекта `friend` является объект `person`, поэтому вызов `super.getGreeting()` эквивалентен вызову `person.getGreeting.call(this)`.

В заключение

Объекты — основа программирования на языке JavaScript, и спецификация ECMAScript 6 внесла несколько интересных изменений в объекты, которые упрощают работу с ними и делают их более гибкими.

Несколько изменений в ECMAScript 6 затрагивают литералы объектов. Сокращенный синтаксис определения свойств упрощает их инициализацию значениями одноименных локальных переменных. Поддержка вычисляемых имен свойств позволяет использовать небуквенные символы, которые допустимы в других конструкциях языка. Сокращенный синтаксис определения методов позволяет сэкономить несколько символов при определении методов в литералах объектов, отбросив двоеточие и ключевое слово `function`. ECMAScript 6 ослабила требования строгого режима, устранив

проверку дубликатов свойств в литералах объектов. Теперь литерал объекта может содержать два определения свойств с одинаковыми именами, не вызывая ошибки.

Новый метод `Object.assign()` упрощает изменение свойств в одном объекте и с успехом может использоваться для реализации шаблона «Примесь». Метод `Object.is()` выполняет строгое сравнение любых значений и фактически является безопасной версией оператора `===`, поддерживающей особые значения JavaScript.

Спецификация ECMAScript 6 ясно определила порядок перечисления собственных свойств объектов. Первыми всегда возвращаются числовые свойства, следующие в порядке возрастания, за ними идут строковые ключи в порядке добавления в объект при его определении, а за ними следуют символические ключи также в порядке добавления в объект.

Теперь благодаря введению в ECMAScript 6 метода `Object.setPrototypeOf()` появилась возможность менять прототип объекта после его создания.

Кроме того, вызов методов прототипа объекта теперь можно осуществлять с помощью ссылки `super`. Привязка `this` внутри метода, вызванного с помощью `super`, автоматически настраивается для работы с текущим объектом.

Деструктуризация для упрощения доступа к данным

Литералы объектов и массивов — две часто используемые формы записи в JavaScript, и благодаря популярности формата данных JSON они стали одной из важнейших частей языка. В программах часто можно увидеть, как определяются объекты и массивы, а затем из них извлекаются те или иные фрагменты информации. ECMAScript 6 упрощает эту задачу, добавив операцию *деструктуризации*, которая осуществляет дробление структуры данных на более мелкие части. Эта глава покажет вам, как применять деструктуризацию к объектам и массивам.

Какие выгоды дает деструктуризация?

В ECMAScript 5 и в более ранних версиях необходимость извлечения информации из объектов и массивов могла повлечь появление большого объема повторяющегося кода, извлекающего определенные данные в локальные переменные. Например:

```
let options = {
  repeat: true,
  save: false
};

// извлечь данные из объекта
let repeat = options.repeat,
    save = options.save;
```

Этот код извлекает значения свойств `repeat` и `save` из объекта `options` и сохраняет данные в одноименных локальных переменных. Хотя этот код выглядит вполне простым, но представьте, что требуется присвоить значения большому количеству переменных — вам придется для каждой написать отдельную инструкцию присваивания. А если потребуется обойти вложенную структуру данных в поисках информации, для этого может понадобиться углубиться в структуру до самого нижнего уровня, только чтобы найти нужные сведения.

Чтобы избавить разработчиков от всех этих сложностей, в ECMAScript 6 была добавлена поддержка деструктуризации объектов и массивов. Она существенно упрощает извлечение информации из структур данных. Многие языки поддерживают деструктуризацию с минимальным синтаксисом, чтобы упростить ее применение. Реализация в ECMAScript 6 использует уже знакомый вам синтаксис — синтаксис литералов объектов и массивов.

Деструктуризация объектов

В синтаксисе деструктуризации объектов слева от оператора присваивания используется литерал объекта. Например:

```
let node = {
  type: "Identifier",
  name: "foo"
};

let { type, name } = node;

console.log(type);    // "Identifier"
console.log(name);    // "foo"
```

Этот код сохранит значение `node.type` в переменной с именем `type`, а значение `node.name` — в переменной с именем `name`. Здесь используется сокращенный синтаксис инициализации свойств в литералах объектов, представленный в главе 4. Идентификаторы `type` и `name` одновременно объявляют локальные переменные и определяют свойства объекта `node` для извлечения.

НЕ ЗАБЫВАЙТЕ ПРО ИНИЦИАЛИЗАЦИЮ

Объявляя переменные `var`, `let` или `const` с использованием синтаксиса деструктуризации, не забывайте инициализировать их (указывать значение после знака «равно»). Все следующие строки кода вызовут синтаксическую ошибку из-за отсутствия инициализирующего значения:

```
// синтаксическая ошибка!
var { type, name };

// синтаксическая ошибка!
let { type, name

// синтаксическая ошибка!
const { type, name };
```

Если объявление `const` всегда требует выполнять инициализацию, даже для простых переменных, то объявления `var` и `let` требуют наличия инициализирующего значения, только когда используются для деструктуризации.

Присваивание с деструктуризацией

В примерах деструктуризации объектов, показанных выше, использовались объявления переменных. Однако деструктуризацию можно выполнять в обычных операциях присваивания. Например, можно изменить значения переменных после их определения, как показано ниже:

```
let node = {
  type: "Identifier",
  name: "foo"
},
type = "Literal",
name = 5;
```



```
// присвоить другие значения, используя деструктуризацию
({ type, name } = node);

console.log(type);    // "Identifier"
console.log(name);    // "foo"
```

В этом примере свойства `type` и `name` литерала объекта инициализируются значениями в момент объявления, а затем две переменные с теми же именами инициализируются другими значениями. В следующей строке используется операция присваивания с деструктуризацией, чтобы записать в переменные значения свойств объекта `node`. Обратите внимание на необходимость заключения операции присваивания с деструктуризацией в круглые скобки. Причина в том, что в противном случае фигурные скобки будут интерпретироваться как блочная инструкция, а блочные инструкции нельзя использовать слева от оператора присваивания. Круглые скобки сигнализируют интерпретатору, что следующие фигурные скобки не являются блочной инструкцией и должны интерпретироваться как выражение, позволяя завершить присваивание.

Выражение присваивания с деструктуризацией возвращает результат вычисления правой стороны (после знака `=`). Это означает, что такие выражения можно использовать везде, где ожидается значение. Например, следующий фрагмент выполняет передачу объекта в функцию:

```
let node = {
  type: "Identifier",
  name: "foo"
},
type = "Literal",
name = 5;

function outputInfo(value) {
  console.log(value === node);    // true
}
outputInfo({ type, name } = node);

console.log(type);    // "Identifier"
console.log(name);    // "foo"
```

В вызов функции `outputInfo()` передается выражение присваивания с деструктуризацией. Это выражение возвращает объект `node`, потому что он находится с правой стороны выражения. Присваивание значений переменным `type` и `name` выполняется как обычно, а `node` передается функции `outputInfo()`.

ПРИМЕЧАНИЕ

Если правая часть выражения присваивания с деструктуризацией (после знака `=`) вычисляется как `null` или `undefined`, генерируется ошибка. Это объясняется тем, что любая попытка прочитать свойство из значения `null` или `undefined` приводит к ошибке времени выполнения.

Значения по умолчанию

Когда в выражении присваивания с деструктуризацией указывается локальная переменная, для которой в объекте отсутствует одноименное свойство, она получает значение `undefined`. Например:

```
let node = {
  type: "Identifier",
```

```
    name: "foo"
  };

let { type, name, value } = node;

console.log(type);    // "Identifier"
console.log(name);    // "foo"
console.log(value);   // undefined
```

В этом фрагменте определяется дополнительная локальная переменная с именем `value` и выполняется попытка присвоить ей значение. Однако в объекте `node` отсутствует одноименное свойство, поэтому, как и ожидалось, переменная получила значение `undefined`.

При необходимости можно определить значение по умолчанию на тот случай, если в объекте не окажется соответствующего свойства. Для этого нужно добавить знак равно (=) после имени переменной и указать значение по умолчанию, например:

```
let node = {
  type: "Identifier",
  name: "foo"
};

let { type, name, value = true } = node;

console.log(type);    // "Identifier"
console.log(name);    // "foo"
console.log(value);   // true
```

В этом примере переменная `value` получила значение по умолчанию `true`. Значение по умолчанию используется, только если в объекте `node` отсутствует соответствующее свойство или оно имеет значение `undefined`. Так как свойство `node.value` отсутствует, переменная `value` получила значение по умолчанию. Это напоминает значения параметров по умолчанию в функциях, о которых рассказывалось в главе 3.

Присваивание локальным переменным с другими именами

До данного момента в каждом примере присваивания с деструктуризацией использовались одноименные локальные переменные и свойства объектов; например, значение свойства `node.type` сохранялось в переменной `type`. Этот прием хорошо подходит для случаев использования одинаковых имен, но как быть, если имена переменных не совпадают с именами свойств? В ECMAScript 6 был добавлен расширенный синтаксис, позволяющий присваивать значения локальным переменным с другими именами, и этот синтаксис выглядит как литерал объекта с обычным способом инициализации свойств. Например:

```
let node = {
  type: "Identifier",
  name: "foo"
};

let { type: localType, name: localName } = node;

console.log(localType); // "Identifier"
console.log(localName); // "foo"
```

Этот пример использует присваивание с деструктуризацией для объявления переменных `localType` и `localName` и присваивания им значений свойств `node.type` и `node.name` соответственно. Синтаксис `type: localType` описывает чтение свойства с именем `type` и сохранение его значения в переменной `localType`. Этот синтаксис фактически являет собой противоположность традиционному синтаксису объявления литералов объектов, в котором имя указывается слева от двоеточия, а значение — справа. В данном случае имя находится справа от двоеточия, а местоположение для чтения значения — слева.

Используя другие имена переменных, также можно определять значения по умолчанию. Знак равно и значение по умолчанию по-прежнему размещаются после имени локальной переменной. Например:

```
let node = {
  type: "Identifier"
};

let { type: localType, name: localName = "bar" } = node;

console.log(localType);    // "Identifier"
console.log(localName);    // "bar"
```

Здесь для переменной `localName` определено значение по умолчанию `"bar"`. Переменная получит именно это значение, потому что нет такого свойства `node.name`.

Теперь вы знаете, как применять операцию деструктуризации к объектам, свойства которых являются элементарными значениями. Однако деструктуризацию можно также применять для извлечения значений из вложенных объектов.

Деструктуризация вложенных объектов

Используя синтаксис, напоминающий синтаксис литералов объектов, можно углубляться во вложенные структуры объектов для извлечения необходимой информации. Например:

```
let node = {
  type: "Identifier",
  name: "foo",
  loc: {
    start: {
      line: 1,
      column: 1
    },
    end: {
      line: 1,
      column: 4
    }
  }
};

let { loc: { start } } = node;

console.log(start.line);    // 1
console.log(start.column);  // 1
```

В этом примере в шаблоне деструктуризации используются фигурные скобки с целью показать, что интерпретатор должен углубиться в свойство `loc` объекта `node` и найти свойство `start`.

Напомним еще раз, что слева от двоеточия в шаблоне деструктуризации указывается идентификатор для исследования, а справа — идентификатор для присваивания значения. Фигурные скобки после двоеточия сообщают, что искомое свойство находится в объекте уровнем ниже.

Здесь также можно использовать локальные переменные с именами, отличающимися от имен свойств объектов:

```
let node = {
  type: "Identifier",
  name: "foo",
  loc: {
    start: {
      line: 1,
      column: 1
    },
    end: {
      line: 1,
      column: 4
    }
  }
};

// извлечь node.loc.start
let { loc: { start: localStart } } = node;

console.log(localStart.line);    // 1
console.log(localStart.column);  // 1
```

В этой версии кода значение `node.loc.start` сохраняется в новой локальной переменной с именем `localStart`. Шаблон деструктуризации может иметь произвольное количество уровней вложенности, и на каждом уровне доступны все имеющиеся возможности.

Деструктуризация объектов — очень мощная особенность, потому что открывает массу возможностей, но деструктуризация массивов предлагает свой ряд уникальных способов извлечения информации из массивов.

СИНТАКСИЧЕСКАЯ ЛОВУШКА

Будьте осторожны, используя синтаксис деструктуризации вложенных элементов, потому что можно по неосторожности создать инструкцию, не оказывающую никакого эффекта. Пустые фигурные скобки допустимы в шаблоне деструктуризации, но они ничего не делают. Например:

```
// переменная не будет создана!
let { loc: {} } = node;
```

Эта инструкция не создает никаких привязок. Благодаря фигурным скобкам справа `loc` используется как местоположение для исследования, а не для создания привязки. В таких случаях создается впечатление, что разработчик намеревался применить `=`, чтобы определить значение по умолчанию, а не определить вместо этого местоположение с помощью `:`. В будущем, возможно, этот синтаксис будет объявлен недопустимым, но пока это ловушка, в которую желательно не попадать.

Деструктуризация массивов

Синтаксис деструктуризации массивов очень напоминает деструктуризацию объектов, только вместо синтаксиса литералов объектов используется синтаксис литералов массивов. Деструктуризация применяется к позициям в массиве, а не к именованным свойствам, доступным в объектах. Например:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, secondColor ] = colors;

console.log(firstColor);    // "red"
console.log(secondColor);   // "green"
```

Здесь посредством деструктуризации из массива `colors` извлекаются значения `"red"` и `"green"` и сохраняются в переменных `firstColor` и `secondColor`. Значения извлекаются по номерам их позиций в массиве, а имена переменных могут быть любыми. Любые элементы, явно не упомянутые в шаблоне деструктуризации, игнорируются. Обратите внимание, что исходный массив при этом никак не изменяется.

В шаблоне деструктуризации допускается пропускать элементы и указывать имена переменных только для элементов, представляющих интерес. Если, к примеру, потребуется извлечь из массива только третий элемент, можно не указывать переменные для сохранения первого и второго элементов. Ниже показано, как это делается:

```
let colors = [ "red", "green", "blue" ];

let [ , , thirdColor ] = colors;

console.log(thirdColor);    // "blue"
```

В этом фрагменте используется операция присваивания с деструктуризацией, извлекающая третий элемент из массива `colors`. Запятые, предшествующие в шаблоне имени переменной `thirdColor`, отмечают поля, соответствующие предшествующим элементам массива. Используя такой прием, легко можно пропустить значения из произвольных элементов в середине массива без необходимости указывать имена переменных.

ПРИМЕЧАНИЕ

По аналогии с деструктуризацией объектов при выполнении деструктуризации массива в операторах `var`, `let` или `const` всегда требуется указывать инициализирующее значение.

Присваивание с деструктуризацией

Деструктуризацию массивов можно использовать в контексте операции присваивания, но, в отличие от деструктуризации объектов, в этом случае нет необходимости заключать выражение в круглые скобки. Взгляните на следующий пример.

```
let colors = [ "red", "green", "blue" ],
    firstColor = "black",
    secondColor = "purple";

[ firstColor, secondColor ] = colors;
```

```
console.log(firstColor);    // "red"
console.log(secondColor);   // "green"
```

Присваивание с деструктуризацией в этом фрагменте действует почти так же, как в предыдущем примере. Единственное отличие — переменные `firstColor` и `secondColor` уже были определены к этому моменту. Сейчас вы знаете практически все, что требуется знать об операции присваивания массивов с деструктуризацией, но есть еще кое-что, что может пригодиться вам в будущем.

Операция присваивания массивов с деструктуризацией обладает уникальным свойством, упрощающим обмен значений между двумя переменными. Обмен переменных значениями часто используется в алгоритмах сортировки, и в ECMAScript 5 для этой цели используется третья временная переменная, как показано в следующем примере:

```
// обмен значений переменных в ECMAScript 5
let a = 1,
    b = 2,
    tmp;

tmp = a;
a = b;
b = tmp;

console.log(a);    // 2
console.log(b);    // 1
```

Чтобы поменять местами значения переменных `a` и `b`, необходима временная переменная `tmp`. Однако операция присваивания массивов с деструктуризацией избавляет от необходимости использовать третью переменную. Ниже показано, как поменять значения переменных в ECMAScript 6:

```
// обмен значений переменных в ECMAScript 6
let a = 1,
    b = 2;

[ a, b ] = [ b, a ];

console.log(a);    // 2
console.log(b);    // 1
```

Присваивание массива с деструктуризацией в этом примере выглядит как зеркальное отражение. Слева от знака «равно» находится шаблон деструктуризации, как и в других примерах деструктуризации массивов. Справа находится литерал массива, временно созданный для обмена значений. Деструктуризация применяется к временному массиву, в первый и второй элементы которого копируются значения из `b` и `a`. В результате возникает эффект обмена переменных значениями.

ПРИМЕЧАНИЕ

По аналогии с операцией присваивания объектов с деструктуризацией, если правая часть выражения присваивания вернет `null` или `undefined`, интерпретатор сгенерирует ошибку.

Значения по умолчанию

Синтаксис присваивания массивов с деструктуризацией поддерживает возможность определения значений по умолчанию для любых элементов массива. Значение по умолчанию используется в случае отсутствия элемента в данной позиции или элемент имеет значение `undefined`. Например:

```
let colors = [ "red" ];

let [ firstColor, secondColor = "green" ] = colors;

console.log(firstColor);    // "red"
console.log(secondColor);   // "green"
```

В этом примере массив `colors` имеет только один элемент, то есть в нем отсутствует элемент, соответствующий переменной `secondColor` в шаблоне деструктуризации. Поэтому переменная `secondColor` получила значение по умолчанию `"green"` вместо `undefined`.

Деструктуризация вложенных массивов

По аналогии с деструктуризацией вложенных объектов поддерживается также возможность деструктуризации вложенных массивов. Добавляя в общий шаблон вложенные шаблоны массивов, можно организовать деструктуризацию вложенных массивов, например:

```
let colors = [ "red", [ "green", "lightgreen" ], "blue" ];

// позднее

let [ firstColor, [ secondColor ] ] = colors;

console.log(firstColor);    // "red"
console.log(secondColor);   // "green"
```

Здесь переменная `secondColor` получила значение `"green"` из массива, вложенного в массив `colors`. Элемент с этим значением содержится во втором массиве, поэтому в шаблоне деструктуризации понадобилось использовать дополнительные квадратные скобки вокруг `secondColor`. Так же как в случае с объектами, глубина вложенности массивов не ограничивается.

Остаточные элементы

В главе 3 были представлены остаточные параметры функций. Синтаксис деструктуризации массивов имеет схожее понятие, которое называется *остаточные элементы (rest items)*. Для обозначения остаточных элементов используется синтаксис `...`, который служит для присваивания оставшихся элементов в массиве конкретной переменной. Взгляните на следующий пример:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, ...restColors ] = colors;

console.log(firstColor);    // "red"
console.log(restColors.length); // 2
console.log(restColors[0]);  // "green"
console.log(restColors[1]);  // "blue"
```

Первый элемент массива `colors` присваивается переменной `firstColor`, а остальные копируются в новый массив `restColors`. В результате `restColors` получает два элемента: `"green"` и `"blue"`. Остаточные элементы удобно использовать для извлечения определенных элементов из массива и сохранения остаточных элементов доступными, однако этот подход имеет еще одно полезное применение.

Большим упущением в поддержке массивов в JavaScript является отсутствие простого способа создания клонов. В ECMAScript 5 разработчики часто используют для этой цели метод `concat()`. Например:

```
// клонирование массивов в ECMAScript 5
var colors = [ "red", "green", "blue" ];
var clonedColors = colors.concat();

console.log(clonedColors);    // "[red,green,blue]"
```

Вообще метод `concat()` предназначен для объединения двух массивов, но если вызвать его без аргумента, он вернет клон массива. В ECMAScript 6 для решения той же задачи можно использовать синтаксис выбора остаточных элементов, как показано ниже:

```
// клонирование массивов в ECMAScript 6
let colors = [ "red", "green", "blue" ];
let [ ...clonedColors ] = colors;

console.log(clonedColors);    // "[red,green,blue]"
```

В этом примере для копирования значений из массива `colors` в массив `clonedColors` использован синтаксис остаточных элементов. Несмотря на то что ясность намерений разработчика, использовавшего этот прием, остается спорной, если сравнивать с приемом на основе метода `concat()`, такая возможность достаточно полезна, чтобы знать о ее существовании.

ПРИМЕЧАНИЕ

Остаточные элементы должны быть последним компонентом в шаблоне деструктуризации массива и за ними не может следовать запятая. Добавление запятой после остаточных элементов вызывает синтаксическую ошибку.

Смешанная деструктуризация

Шаблоны деструктуризации объектов и массивов можно смешивать для создания более сложных выражений. Используя этот прием, можно извлекать только требуемые фрагменты информации из смеси объектов и массивов. Взгляните на следующий пример:

```
let node = {
  type: "Identifier",
  name: "foo",
  loc: {
    start: {
      line: 1,
      column: 1
    },
    end: {
      line: 1,
      column: 4
    }
  },
  range: [0, 3]
};

let {
  loc: { start },
  range: [ startIndex ]
} = node;
```



```
console.log(start.line);      // 1
console.log(start.column);    // 1
console.log(startIndex);      // 0
```

Этот фрагмент извлекает значения `node.loc.start` и `node.range[0]` в переменные `start` и `startIndex` соответственно. Обратите внимание, что `loc:` и `range:` в шаблоне деструктуризации — это лишь местоположения, соответствующие свойствам объекта `node`. В объекте `node` нет ничего, что нельзя было бы извлечь с использованием смешанного синтаксиса деструктуризации массивов и объектов. Такой подход особенно удобно применять для извлечения значений из конфигурационных структур в формате JSON без необходимости выполнять обход всей структуры.

Деструктурированные параметры

Деструктуризация имеет еще одну область применения, где она оказывается особенно удобной, — это передача аргументов функциям. Когда функция принимает большое количество необязательных аргументов, разработчики часто создают объект `options` со свойствами, определяющими необязательные параметры. Например:

```
// свойства в options представляют дополнительные параметры
function setCookie(name, value, options) {

    options = options || {};

    let secure = options.secure,
        path = options.path,
        domain = options.domain,
        expires = options.expires;

    // код, настраивающий cookie
}

// третий аргумент отображается в параметр options
setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

Многие библиотеки на JavaScript содержат функцию `setCookie()`, похожую на представленную выше. В этой функции параметры `name` и `value` являются обязательными, но `secure`, `path`, `domain` и `expires` — нет. А так как нет предпочтительного порядка передачи других данных, эффективнее передавать объект `options` с именованными свойствами, чем перечислять дополнительные именованные параметры. Это вполне работоспособное решение, но при беглом взгляде на такое определение функции нельзя сказать, какие данные она ожидает, — вам придется изучить ее тело.

Деструктурированные параметры предлагают альтернативное решение, помогающее ясно показать, какие параметры ожидаются функцией. В определении деструктурированных параметров на месте именованного параметра используется шаблон деструктуризации объекта или массива. Чтобы увидеть, как применяется этот прием, взгляните на переписанную версию функции `setCookie()` из предыдущего примера:

```
function setCookie(name, value, { secure, path, domain, expires }) {
```

```
// код, настраивающий cookie
}

setCookie("type", "js", {
  secure: true,
  expires: 60000
});
```

Эта функция действует подобно предыдущей, но теперь в третьем параметре используется шаблон деструктуризации, описывающий необходимые данные. Параметры, находящиеся за границами шаблона, очевидно, являются обязательными, и также очевидно, какие дополнительные параметры принимает функция `setCookie()`. Кроме того, деструктурированные параметры действуют подобно обычным параметрам, и в случае их отсутствия в вызове функции они получают значение `undefined`.

ПРИМЕЧАНИЕ

Деструктурированные параметры обладают всеми свойствами деструктуризации, с которыми вы познакомились в этой главе. В них можно использовать значения по умолчанию, смешивать шаблоны деструктуризации объектов и массивов и использовать имена переменных, отличающиеся от имен свойств передаваемых объектов.

Деструктурированные параметры являются обязательными

Один из недостатков деструктурированных параметров заключается в том, что в случае их отсутствия в вызове функции по умолчанию возбуждается ошибка. Например, следующий вызов функции `setCookie()` из предыдущего примера приведет к ошибке.

```
// ошибка!
setCookie("type", "js");
```

Отсутствующему третьему аргументу как обычно будет присвоено значение `undefined`, что вызовет ошибку, потому что деструктурированные параметры — это всего лишь деструктурированное объявление. Когда программа выполнит такой вызов `setCookie()`, движок JavaScript фактически сделает следующее:

```
function setCookie(name, value, options) {

  let { secure, path, domain, expires } = options;

  // код, настраивающий cookie
}
```

Так как операция деструктуризации возбуждает ошибку, если правая часть выражения возвращает `null` или `undefined`, эта же ошибка появится, если вызвать функцию `setCookie()` без третьего аргумента.

Если вы хотите сделать деструктурированный параметр обязательным, такое поведение не является проблемой. Но если требуется сделать деструктурированный параметр необязательным, эту задачу можно решить, определив значение по умолчанию, как показано ниже:

```
function setCookie(name, value, { secure, path, domain, expires } = {}) {
  // пустая
}
```

В этом примере третий параметр имеет значение по умолчанию — пустой объект. Такое значение по умолчанию для деструктурированного параметра означает, что `secure`, `path`, `domain` и `expires` получают значение `undefined`, если функция `setCookie()` будет вызвана без третьего аргумента, и никаких ошибок при этом не возникнет.

Значения по умолчанию для деструктурированных параметров

Для деструктурированных параметров допускается определять деструктурированные значения по умолчанию, как это делается в операции присваивания с деструктуризацией. Достаточно просто добавить знак «равно» после параметра и указать значение по умолчанию. Например:

```
function setCookie(name, value,
  {
    secure = false,
    path = "/",
    domain = "example.com",
    expires = new Date(Date.now() + 3600000000)
  } = {}) {
  // пустая
}
```

В этом примере все свойства в деструктурированном параметре получают значения по умолчанию, поэтому из реализации можно исключить проверку наличия свойства, чтобы подставить правильное значение. Кроме того, значение по умолчанию определено также для всего деструктурированного параметра — пустой объект, что делает его необязательным. Такое объявление функции выглядит немного сложнее, но это невысокая цена за гарантию, что каждый аргумент получит значение, пригодное для использования.

В заключение

Деструктуризация упрощает работу с объектами и массивами в JavaScript. Используя знакомый синтаксис литералов объектов и массивов, вы можете извлекать из структур данных любую необходимую информацию. Шаблоны объектов позволяют извлекать данные из объектов, а шаблоны массивов — из массивов.

В шаблонах обоих видов — объектов и массивов — можно указывать значения по умолчанию для любых свойств или элементов, и в обоих случаях возбуждается ошибка, если правая часть выражения присваивания возвращает `null` или `undefined`. Используя деструктуризацию объектов и массивов, можно извлекать данные из внутренних структур данных, находящихся на произвольной глубине вложенности.

Деструктурированные объявления `var`, `let` и `const`, создающие переменные, всегда должны содержать инициализирующие значения. Присваивание с деструктуризацией используется взамен обычного присваивания с целью присвоить значения свойств объекта уже имеющимся переменным.

Деструктурированные параметры применяют синтаксис деструктуризации, чтобы сделать более прозрачными объекты `options`, используемые в виде параметров функций. Вы можете перечислить все фактические данные, представляющие интерес, наряду с другими именованными параметрами. Деструктурированные параметры можно определять с применением шаблонов массивов, шаблонов объектов или их комбинаций, и использовать в них все свойства механизма деструктуризации.

Символы и символьные свойства

В ECMAScript 6 появилась поддержка символов (symbols) как элементарного типа. (В языке уже имеется пять элементарных типов: строки, числа, логические значения, `null` и `undefined`.) Изначально символы предназначались для создания приватных членов объектов, появления которых так долго ждали разработчики на JavaScript. Пока символы не поддерживались, любые свойства со строковыми именами были легкодоступны независимо от ухищрений по сокрытию их имен, и поддержка *приватных имен* имела целью дать разработчикам возможность создавать свойства с именами, не являющимися строками. В этом случае обычные приемы перечисления приватных имен оказываются бессильными.

Идея создания приватных имен в конечном итоге была воплощена спецификацией ECMAScript 6 в виде символов, и эта глава научит вас эффективно пользоваться ими. Несмотря на то что символы действительно позволяют использовать нестроковые значения в качестве имен свойств, основная их цель — приватность — была утрачена. Взамен символьные свойства образовали категорию свойств, отдельную от других свойств объектов.

Создание символов

Символы занимают уникальное положение среди других примитивов JavaScript — они не имеют литеральной формы, как `true` для логических значений или `42` для чисел. Символ можно создать с помощью глобальной функции `Symbol`, например:

```
let firstName = Symbol();
let person = {};

person[firstName] = "Nicholas";
console.log(person[firstName]);    // "Nicholas"
```

Здесь создается символ `firstName` и используется для добавления нового свойства в объект `person`. Когда в операции присваивания свойству значения применяется символ, этот символ придется использовать при любом обращении к такому свойству. Всегда выбирайте для символов такие имена, чтобы легко можно было сказать, что этот символ представляет.

ПРИМЕЧАНИЕ

Поскольку символы являются элементарными значениями, вызов `new Symbol()` вызовет ошибку. Экземпляр `Symbol` можно также создать вызовом `new Object(yourSymbol)`, но я не знаю, где такой способ мог бы пригодиться.

Функция `Symbol` принимает дополнительный аргумент с описанием символа. Это описание нельзя использовать для доступа к свойству, но я советую всегда добавлять описание, чтобы упростить чтение и отладку символов. Например:

```
let firstName = Symbol("first name");
let person = {};

person[firstName] = "Nicholas";

console.log("first name" in person);    // false
console.log(person[firstName]);         // "Nicholas"
console.log(firstName);                 // "Symbol(first name)"
```

Описание символа хранится во внутреннем свойстве `[[Description]]`. Это свойство читается всякий раз, когда явно или неявно вызывается метод `toString()` символа. В этом примере `console.log()` неявно вызывает метод `toString()` символа `firstName`, поэтому в консоль выводится строка с его описанием. Нет другого способа осуществить доступ к свойству `[[Description]]` из программного кода.

ИДЕНТИФИКАЦИЯ СИМВОЛОВ

Поскольку символы – это элементарные значения, можно применить оператор `typeof` с целью определить, хранит ли некоторая переменная символ. В ECMAScript 6 оператор `typeof` возвращает `"symbol"`, когда применяется к символу. Например:

```
let symbol= Symbol("test symbol");
console.log(typeof symbol);    // "symbol"
```

Хотя есть и другие косвенные способы, позволяющие определить, является ли переменная символом, однако оператор `typeof` считается самым точным и потому наиболее предпочтительным способом.

Использование символов

Символы можно использовать везде, где допустимы вычисляемые имена свойств. В этой главе вы уже видели форму применения символов в квадратных скобках, но кроме этого символы можно использовать в вычисляемых именах свойств в литералах объектов, а также передавать в вызовы `Object.defineProperty()` и `Object.defineProperties()`:

```
let firstName = Symbol("first name");

// использовать в качестве вычисляемого
// имени свойства в литерале объекта
let person = {
  [firstName]: "Nicholas"
};

// сделать свойство доступным только для чтения
Object.defineProperty(person, firstName, { writable: false });

let lastName = Symbol("last name");
```

```
Object.defineProperty(person, {
  [lastName]: {
    value: "Zakas",
    writable: false
  }
});

console.log(person[firstName]); // "Nicholas"
console.log(person[lastName]); // "Zakas"
```

В этом примере символ сначала используется в качестве вычисляемого имени свойства в литерале объекта для создания символьного свойства `firstName`. Свойство создается как неперечислимое, что отличает его от обычных свойств со строковыми именами. В следующей строке свойство превращается в доступное только для чтения. Затем вызовом метода `Object.defineProperty()` создается символьное свойство `lastName`, доступное только для чтения. Здесь снова используется символьное свойство с вычисляемым именем в литерале объекта, но на этот раз оно является частью второго аргумента в вызове `Object.defineProperty()`.

Несмотря на то что символы можно применять везде, где допустимы вычисляемые имена свойств, необходимо предусмотреть систему совместного использования этих символов различными фрагментами кода для эффективной работы с ними.

Совместное использование символов

Иногда может возникнуть необходимость совместного использования символов различными частями вашего кода. Например, допустим, что в приложении имеется два разных типа объектов, которые должны использовать одно и то же символьное свойство для представления уникального идентификатора. Уследить за всеми символами во множестве файлов или в большой базе программного кода — не самая простая задача, и высока вероятность ошибки. По этой причине в ECMAScript 6 предусмотрен глобальный реестр символов, к которому можно обратиться в любой момент.

При необходимости создать символ для общего пользования используйте метод `Symbol.for()` вместо `Symbol()`. Метод `Symbol.for()` принимает единственный параметр — строку, идентифицирующую создаваемый символ. Этот параметр также применяется как описание символа, например:

```
let uid = Symbol.for("uid");
let object = {};

object[uid] = "12345";

console.log(object[uid]); // "12345"
console.log(uid); // "Symbol(uid)"
```

Метод `Symbol.for()` сначала просмотрит глобальный реестр символов, пытаясь найти символ с ключом `"uid"`. Если такой символ будет найден, метод вернет существующий символ. В противном случае метод создаст новый символ, зарегистрирует его в глобальном реестре с указанным ключом и вернет вызывающей программе.

Последующие попытки вызвать метод `Symbol.for()` с тем же ключом будут возвращать тот же самый символ, как показано ниже:

```
let uid = Symbol.for("uid");
let object = {
  [uid]: "12345"
```

```
};

console.log(object[uid]);    // "12345"
console.log(uid);           // "Symbol(uid)"

let uid2 = Symbol.for("uid");

console.log(uid === uid2);   // true
console.log(object[uid2]);   // "12345"
console.log(uid2);           // "Symbol(uid)"
```

В этом примере переменные `uid` и `uid2` хранят один и тот же символ и могут использоваться взаимозаменяемо. Первый вызов `Symbol.for()` создает символ, а второй — вызов извлекает символ из глобального реестра.

Другая уникальная особенность совместно используемых символов заключается в возможности извлекать ключ, связанный с символом в глобальном реестре, вызовом метода `Symbol.keyFor()`. Например:

```
let uid = Symbol.for("uid");
console.log(Symbol.keyFor(uid));    // "uid"

let uid2 = Symbol.for("uid");
console.log(Symbol.keyFor(uid2));    // "uid"

let uid3 = Symbol("uid");
console.log(Symbol.keyFor(uid3));    // undefined
```

Обратите внимание, что обеим переменным — `uid` и `uid2` — присваивается ключ `"uid"`. Символ в переменной `uid3` не был включен в глобальный реестр, поэтому для него метод `Symbol.keyFor()` вернул значение `undefined`.

ПРИМЕЧАНИЕ

Глобальный реестр символов является совместно используемым окружением, в точности как глобальная область видимости. Это означает, что нельзя делать никаких предположений о том, что уже имеется или отсутствует в этом окружении. Применяйте пространства имен для ключей символов, чтобы уменьшить вероятность конфликтов при использовании сторонних компонентов. Например, в jQuery ко всем ключам символов может добавляться префикс `"jquery."`, например `"jquery.element"`.

Приведение типов для символов

Приведение типов — важная особенность JavaScript, а гибкость языка во многом зависит от возможности преобразования одних типов данных в другие. Однако символы — весьма негибкий тип данных, потому что в других типах отсутствуют логичные эквиваленты символов. В частности, символы нельзя преобразовать в строки или числа, чтобы предотвратить случайное их использование в качестве свойств там, где ожидаются символы.

Примеры в этой главе используют функцию `console.log()`, потому что для вывода символов она вызывает функцию `String()`, возвращающую удобочитаемую строку. Вы можете использовать `String()` напрямую, чтобы получить тот же результат. Например:

```
let uid = Symbol.for("uid"),
    desc = String(uid);
```

```
console.log(desc);    // "Symbol(uid)"
```

Функция `String()` вызывает метод `uid.toString()`, возвращающий строку с описанием символа. Однако попытка объединить символ со строкой вызовет ошибку:

```
var uid = Symbol.for("uid"),  
    desc = uid + "";    // ошибка!
```

Операция конкатенации `uid` с пустой строкой требует преобразовать `uid` в строку и когда интерпретатор сталкивается с такой необходимостью, он генерирует ошибку, чтобы воспрепятствовать такому использованию символа.

Попытка преобразовать символ в число также завершается ошибкой. Все математические операции вызывают ошибку, когда обнаруживается, что они применяются к символу. Например:

```
var uid = Symbol.for("uid"),  
    sum = uid / 1;    // ошибка!
```

В этом примере предпринимается попытка разделить символ на 1, что вызывает ошибку. Ошибки возбуждаются независимо от вида математической операции (логические операторы не вызывают ошибок, потому что все символы считаются эквивалентными значению `true`, как любое другое непустое значение в JavaScript).

Извлечение символьных свойств

С помощью методов `Object.keys()` и `Object.getOwnPropertyNames()` можно получить имена всех свойств заданного объекта. Первый метод возвращает имена всех перечислимых свойств, а второй — имена вообще всех свойств, перечислимых и неперечислимых. Однако ни один из них не возвращает символьных свойств, чтобы сохранить их функциональную совместимость с ECMAScript 5. По этой причине в ECMAScript 6 был добавлен метод `Object.getOwnPropertySymbols()`, позволяющий извлекать символьные свойства объектов.

Метод `Object.getOwnPropertySymbols()` возвращает массив собственных символьных свойств, как показано ниже:

```
let uid = Symbol.for("uid");  
let object = {  
    [uid]: "12345"  
};  
  
let symbols = Object.getOwnPropertySymbols(object);  
  
console.log(symbols.length);    // 1  
console.log(symbols[0]);        // "Symbol(uid)"  
console.log(object[symbols[0]]); // "12345"
```

В этом примере `object` имеет единственное символьное свойство с именем `uid`. Метод `Object.getOwnPropertySymbols()` возвращает массив, содержащий только этот символ.

Все объекты изначально не имеют собственных символьных свойств, но могут наследовать символьные свойства от своих прототипов. В ECMAScript 6 определяется несколько таких свойств, реализованных с использованием *стандартных символов*.

Экспортирование внутренних операций в виде стандартных символов

Главной темой для ECMAScript 5 было экспортирование и определение некоторых «необычных» возможностей JavaScript, которые разработчики не могли симитировать в то время. Спецификация ECMAScript 6 продолжила эту традицию и предусматривает экспортирование еще большего количества возможностей, прежде принадлежавших внутренней логике языка, для определения основ поведения некоторых объектов в основном за счет применения прототипа символьных свойств.

В ECMAScript 6 имеется несколько предопределенных символов, которые называются *стандартными символами* (*well-known symbols*) и представляют типовые операции JavaScript, прежде считавшиеся внутренними. Каждый стандартный символ представлен свойством объекта `Symbol`, например `Symbol.match`.

К стандартным символам относятся:

- `Symbol.hasInstance`. Метод, используется оператором `instanceof` для проверки принадлежности к классу.
- `Symbol.isConcatSpreadable`. Логическое значение, указывающее, что метод `Array.prototype.concat()` должен разбивать массивы на отдельные элементы, получаемые в аргументах.
- `Symbol.iterator`. Метод, возвращающий итератор (описывается в главе 8).
- `Symbol.match`. Метод, используемый `String.prototype.match()` для сравнения строк.
- `Symbol.replace`. Метод, используемый `String.prototype.replace()` для замены подстрок.
- `Symbol.search`. Метод, используемый `String.prototype.search()` для поиска подстрок.
- `Symbol.species`. Конструктор для создания производных классов (описывается в главе 9).
- `Symbol.split`. Метод, используемый `String.prototype.split()` для разбиения строк.
- `Symbol.toPrimitive`. Метод, возвращающий элементарное представление значения объекта.
- `Symbol.toStringTag`. Строка, используемая `Object.prototype.toString()` для создания описания объекта.
- `Symbol.unscopables`. Объект, свойства которого являются именами свойств объектов, не подлежащих включению в инструкцию `with`.

Некоторые из стандартных символов обсуждаются в следующих разделах, другие будут обсуждаться на протяжении оставшейся части книги, там, где это уместно.

Перезапись метода, определяемого стандартным символом, превращает обычный объект в экзотический, потому что при этом изменяется некоторое внутреннее поведение по умолчанию. Этот прием не оказывает практического влияния на ваш код; судя по описанию в спецификации, он просто изменяет объект.

Метод `Symbol.hasInstance`

Каждая функция имеет метод `Symbol.hasInstance`, определяющий, является ли данный объект экземпляром этой функции. Метод определяется в `Function.prototype`, поэтому все функции наследуют поведение по умолчанию для свойства `instanceof`. Свойство `Symbol.hasInstance` определено как недоступное для записи, настройки и перечисления, чтобы гарантировать невозможность его перезаписи по ошибке.

Метод `Symbol.hasInstance` принимает единственный аргумент: проверяемое значение. Он возвращает `true`, если указанное значение является экземпляром функции. Чтобы проще было понять, как действует метод `Symbol.hasInstance`, взгляните на следующую инструкцию:

```
obj instanceof Array;
```

Она эквивалентна инструкции:

```
Array[Symbol.hasInstance](obj);
```

Фактически ECMAScript 6 определяет оператор `instanceof` как сокращенный вариант вызова метода. И теперь, когда в работе участвует вызов метода, вы можете изменить поведение оператора `instanceof`.

Например, допустим, что требуется определить функцию, которая утверждает, что никакой объект не является ее экземпляром. Для этого достаточно заставить `Symbol.hasInstance` вернуть `false`, как показано ниже:

```
function MyObject() {  
    // пустая  
}  
  
Object.defineProperty(MyObject, Symbol.hasInstance, {  
    value: function(v) {  
        return false;  
    }  
});  
  
let obj = new MyObject();  
  
console.log(obj instanceof MyObject);    // false
```

Чтобы перезаписать свойство, недоступное для записи, необходимо применить метод `Object.defineProperty()`, как это делается в данном примере, использующем этот метод для перезаписи метода `Symbol.hasInstance` новой функцией. Новая функция всегда возвращает `false`, поэтому, даже если `obj` действительно является экземпляром класса `MyObject`, оператор `instanceof` вернет `false` после вызова `Object.defineProperty()`.

Конечно, можно также проверить значение и исходя из некоторых условий решить, должно ли значение считаться экземпляром. Например, числовые значения от 1 до 100 можно считать экземплярами специального числового типа. Чтобы получить это поведение, можно написать такой код:

```
function SpecialNumber() {  
    // пустая  
}  
  
Object.defineProperty(SpecialNumber, Symbol.hasInstance, {  
    value: function(v) {
```

```
        return (v instanceof Number) && (v >= 1 && v <= 100);
    }
});
```

```
var two = new Number(2),
    zero = new Number(0);
```

```
console.log(two instanceof SpecialNumber);    // true
console.log(zero instanceof SpecialNumber);    // false
```

Этот код определяет метод `Symbol.hasInstance`, возвращающий `true`, если значение является экземпляром `Number`, а также находится в границах между 1 и 100. Поэтому `SpecialNumber` утверждает, что `two` является его экземпляром, даже если между функцией `SpecialNumber` и переменной `two` не существует прямой связи. Обратите внимание, что левый операнд `instanceof` должен быть объектом, чтобы произошел вызов `Symbol.hasInstance`, потому что для значений, не являющихся объектами, оператор `instanceof` всегда возвращает просто `false`.

ПРИМЕЧАНИЕ

Можно также перезаписать свойство по умолчанию `Symbol.hasInstance` во всех встроенных функциях, таких как `Date` и `Error`. Однако это не рекомендуется, потому что влияние на код может оказаться весьма неожиданным. Лучше переопределять метод `Symbol.hasInstance` только для собственных функций, и только когда это действительно необходимо.

Свойство `Symbol.isConcatSpreadable`

Метод `concat()` массивов в JavaScript предназначен для слияния двух массивов. Ниже показано, как он используется:

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ]);

console.log(colors2.length);    // 4
console.log(colors2);           // ["red","green","blue","black"]
```

Этот код добавляет новый массив в конец `colors1` и создает `colors2` — единый массив, включающий все элементы из обоих массивов. Но метод `concat()` может также принимать аргументы, не являющиеся массивами; такие аргументы просто добавляются в конец массива. Например:

```
let colors1 = [ "red", "green" ],
    colors2 = colors1.concat([ "blue", "black" ], "brown");

console.log(colors2.length);    // 5
console.log(colors2);           // ["red","green","blue","black","brown"]
```

Здесь в вызов `concat()` передается дополнительный аргумент `"brown"`, который становится пятым элементом в массиве `colors2`. Почему аргумент-массив обрабатывается иначе, чем строковый аргумент? В спецификации языка JavaScript говорится, что массивы автоматически разбиваются на отдельные элементы, а другие типы — нет. До спецификации ECMAScript 6 не было никакой возможности скорректировать это поведение.

Свойство `Symbol.isConcatSpreadable` — это логическое значение, которое указывает, что если объект имеет свойство `length` и числовые ключи, значения его числовых свойств должны

добавляться в результат вызова `concat()` по отдельности. В отличие от других стандартных символов, это символьное свойство по умолчанию присутствует не у всех стандартных объектов. Оно дает возможность повлиять на поведение по умолчанию метода `concat()` определенных типов объектов. Вы можете заставить любой тип действовать подобно массивам, вызывая метод `concat()`, как показано ниже:

```
let collection = {
  0: "Hello",
  1: "world",
  length: 2,
  [Symbol.isConcatSpreadable]: true
};

let messages = [ "Hi" ].concat(collection);

console.log(messages.length);    // 3
console.log(messages);          // ["hi", "Hello", "world"]
```

Объект `collection` в этом примере настроен так, что выглядит как массив: он имеет свойство `length` и два числовых ключа. Свойству `Symbol.isConcatSpreadable` присвоено значение `true` с целью показать, что значения свойств объекта должны добавляться в массив по отдельности. Когда `concat()` получает объект `collection`, он помещает в массив результата `"Hello"` и `"world"` в виде отдельных элементов, следующих за элементом `"hi"`.

ПРИМЕЧАНИЕ

Свойству `Symbol.isConcatSpreadable` можно также присвоить значение `false` в классах, производных от массивов, чтобы предотвратить их деление на элементы в вызовах `concat()`. Подробности смотрите в разделе «Наследование в производных классах» главы 9.

Свойства `Symbol.match`, `Symbol.replace`, `Symbol.search` и `Symbol.split`

Строки и регулярные выражения всегда были тесно связаны между собой в JavaScript. В частности, строковый тип имеет несколько методов, принимающих регулярные выражения в качестве аргументов:

- `match(regex)`. Определяет, соответствует ли данная строка регулярному выражению.
- `replace(regex, replacement)`. Замещает совпадения с регулярным выражением строкой `replacement`.
- `search(regex)`. Отыскивает в строке совпадение с регулярным выражением.
- `split(regex)`. Преобразует строку в массив, разбивая ее по совпадениям с регулярным выражением.

До появления ECMAScript 6 взаимодействия этих методов с регулярными выражениями были скрыты от разработчиков, что лишало их возможности имитировать регулярные выражения с использованием своих объектов. В ECMAScript 6 определяются четыре символа, соответствующие этим четырем методам, фактически экспортирующие предопределенное поведение встроенного объекта `RegExp`.

Символы `Symbol.match`, `Symbol.replace`, `Symbol.search` и `Symbol.split` представляют методы аргумента с регулярным выражением, которые должны вызываться для первого аргумента

метода `match()`, метода `replace()`, метода `search()` и метода `split()` соответственно. Четыре символьных свойства принадлежат `RegExp.prototype` и определяют реализацию по умолчанию, которая должна использоваться строковыми методами.

Зная это, можно создать объект для применения в строковых методах под видом регулярного выражения. Для этого можно использовать следующие символьные функции:

- `Symbol.match`. Принимает строковый аргумент и возвращает массив совпадений или `null` в случае их отсутствия.
- `Symbol.replace`. Принимает строковый аргумент и строку замены и возвращает строку.
- `Symbol.search`. Принимает строковый аргумент и возвращает числовой индекс совпадения или `-1`, если совпадение не найдено.
- `Symbol.split`. Принимает строковый аргумент и возвращает массив с фрагментами исходной строки, разбитой по совпадениям.

Возможность определять эти свойства в объектах позволяет создавать объекты, реализующие сопоставление с шаблоном без применения регулярных выражений, и использовать эти объекты в методах, принимающих регулярные выражения. Следующий пример демонстрирует эти символы в действии:

```
// фактический эквивалент регулярного выражения /^.{10}$/
let hasLengthOf10 = {
  [Symbol.match]: function(value) {
    return value.length === 10 ? [value.substring(0, 10)] : null;
  },
  [Symbol.replace]: function(value, replacement) {
    return value.length === 10 ? replacement + value.substring(10) : value;
  },
  [Symbol.search]: function(value) {
    return value.length === 10 ? 0 : -1;
  },
  [Symbol.split]: function(value) {
    return value.length === 10 ? ["", ""] : [value];
  }
};

let message1 = "Hello world",    // 11 знаков
    message2 = "Hello John";    // 10 знаков

let match1 = message1.match(hasLengthOf10),
    match2 = message2.match(hasLengthOf10);

console.log(match1);             // null
console.log(match2);             // ["Hello John"]

let replace1 = message1.replace(hasLengthOf10),
    replace2 = message2.replace(hasLengthOf10);

console.log(replace1);           // "Hello world"
console.log(replace2);           // "Hello John"

let search1 = message1.search(hasLengthOf10),
```

```
search2 = message2.search(hasLengthOf10);

console.log(search1);           // -1
console.log(search2);           // 0

let split1 = message1.split(hasLengthOf10),
    split2 = message2.split(hasLengthOf10);

console.log(split1);             // ["Hello world"]
console.log(split2);             // ["", ""]
```

Цель создания объекта `hasLengthOf10` — получить эквивалент регулярного выражения, которому соответствуют строки, имеющую длину, равную 10. Все четыре метода в `hasLengthOf10` реализуются с использованием соответствующих символов и затем применяются к двум строкам. Первая строка, `message1`, содержит 11 знаков и не соответствует условию; вторая строка, `message2`, содержит 10 знаков и соответствует условию. Несмотря на то что объект `hasLengthOf10` не является регулярным выражением, он передается в вызовы строковых методов и правильно используется ими благодаря дополнительным методам.

Это был лишь простой пример, тем не менее возможность выполнять более сложные сопоставления, чем позволяют регулярные выражения, открывает широкое поле для создания нестандартных средств сопоставления с шаблоном.

Метод `Symbol.toPrimitive`

JavaScript часто пытается неявно преобразовывать объекты в элементарные значения, когда применяются определенные операции. Например, при сравнении строки с объектом с использованием оператора равенства (`==`) перед сравнением объект преобразуется в элементарное значение. Каким должно быть это элементарное значение, раньше определялось внутренними операциями, но ECMAScript 6 открыла доступ к этому механизму (сделав его изменяемым) в виде метода `Symbol.toPrimitive`.

Метод `Symbol.toPrimitive` определяется в прототипе каждого стандартного типа и реализует преобразование объектов в элементарные значения. Когда возникает необходимость в таком преобразовании, вызывается метод `Symbol.toPrimitive` с единственным аргументом, который в спецификации упоминается под именем `hint`. Аргумент `hint` может принимать одно из трех строковых значений. Если в нем передается строка `"number"`, метод `Symbol.toPrimitive` должен вернуть число. При передаче строки `"string"` должна быть возвращена строка, а передача строки `"default"` означает, что операция не имеет каких-то предпочтений в отношении типа.

Для большинства стандартных объектов числовой режим действует, как описывается ниже:

1. Вызывается метод `valueOf()` и возвращается его результат, если он является элементарным значением.
2. Иначе вызывается метод `toString()` и возвращается его результат, если он является элементарным значением.
3. Иначе возбуждается ошибка.

Аналогично для большинства стандартных объектов строковый режим действует по следующему алгоритму:

1. Вызывается метод `toString()` и возвращается его результат, если он является элементарным значением.
2. Иначе вызывается метод `valueOf()` и возвращается его результат, если он является элементарным значением.

3. Иначе возбуждается ошибка.

Во многих случаях стандартные объекты интерпретируют режим по умолчанию ("default") как эквивалент числового режима (кроме `Date`, для которого по умолчанию действует строковый режим). Определив свой метод `Symbol.toPrimitive`, можно изменить порядок преобразования, установленный по умолчанию.

ПРИМЕЧАНИЕ

Режим по умолчанию используется только операторами `==` и `+`, а также когда конструктору `Date` передается единственный аргумент. Большинство операций используют строковый или числовой режим.

Чтобы изменить поведение по умолчанию механизма преобразования, нужно присвоить `Symbol.toPrimitive` функцию как значение. Например:

```
function Temperature(degrees) {
    this.degrees = degrees;
}

Temperature.prototype[Symbol.toPrimitive] = function(hint) {

    switch (hint) {
        case "string":
            return this.degrees + "\u00b0";    // знак градуса

        case "number":
            return this.degrees;

        case "default":
            return this.degrees + " degrees";
    }
};

var freezing = new Temperature(32);

console.log(freezing + "!");    // "32 градуса!"
console.log(freezing / 2);      // 16
console.log(String(freezing));  // "32°"
```

В этом сценарии определяется конструктор `Temperature` и переопределяется метод `Symbol.toPrimitive` в прототипе. Он возвращает разные значения в зависимости от аргумента `hint`, определяющего режим: строковый, числовой или по умолчанию (значение аргумента `hint` выбирается движком JavaScript). В строковом режиме функция `Temperature()` возвращает температуру со значком Юникода, обозначающим градусы. В числовом режиме она возвращает простое число, а в режиме по умолчанию добавляет к числу слово *degrees* (градуса).

Каждая инструкция вывода вызывает передачу аргумента `hint`, отличающегося от других. Оператор `+` включает режим по умолчанию, передавая в аргументе `hint` значение `"default"`, оператор `/` устанавливает числовой режим, передавая в аргументе `hint` значение `"number"`, а функция `String()` устанавливает строковый режим, передавая в аргументе `hint` значение `"string"`. Во всех трех режимах можно возвращать разные значения, но режим по умолчанию действует как строковый или числовой режим.

Свойство `Symbol.toStringTag`

Одной из самых интересных проблем в JavaScript было существование нескольких глобальных окружений выполнения. Такое случалось в веб-браузерах, когда страница имела плавающие фреймы (`iframe`), потому что страница и плавающие фреймы имели собственные окружения выполнения. Обычно это не представляет большой проблемы, потому что передача данных между окружениями выполняется достаточно просто. Проблема возникает при попытке определить тип объекта после его передачи между разными объектами.

Каноническим примером этой проблемы может служить передача массива из фрейма в страницу, содержащую фрейм, или наоборот. В терминологии ECMAScript 6 плавающий фрейм и содержащая его страница представляют разные *пространства (realm)* — окружения выполнения для JavaScript. Каждое пространство имеет собственную глобальную область видимости с собственными копиями глобальных объектов. В каком бы пространстве ни создавался массив, он определенно остается массивом. Однако после передачи массива в другое пространство оператор `instanceof Array` для него будет возвращать `false`, потому что этот массив создавался конструктором из другого пространства, тогда как идентификатор `Array` представляет конструктор из текущего пространства.

Решение проблемы идентификации

Столкнувшись с проблемой идентификации массивов, разработчики быстро нашли хороший способ ее решения. Они выяснили, что вызов стандартного метода `toString()` объекта всегда возвращает предсказуемую строку. В результате во многих библиотеках на JavaScript появилась следующая функция:

```
function isArray(value) {  
    return Object.prototype.toString.call(value) === "[object Array]";  
}  
  
console.log(isArray([]));    // true
```

Несмотря на то что такое решение выглядит немного запутанным, оно хорошо справляется с задачей идентификации массивов во всех браузерах. Метод `toString()` не подходит для идентификации массивов, потому что возвращает строковое представление элементов, содержащихся в объекте. Но метод `toString()` в `Object.prototype` имеет одну особенность: он включает в результат строку, содержащуюся во внутреннем свойстве `[[Class]]`. Благодаря этой особенности разработчики смогли реализовать способ определения типа объекта, созданного в другом окружении.

Разработчики также быстро заметили, что из-за невозможности изменить такое поведение метода тот же подход можно применить, чтобы провести различия между встроенными объектами и объектами, созданными разработчиками. Наиболее важным случаем применения этого решения был объект `JSON` из ECMAScript 5.

До появления ECMAScript 5 многие разработчики использовали библиотеку `json2.js` Дугласа Крокфорда (Douglas Crockford), которая создает глобальный объект `JSON`. Как только в браузерах стала появляться встроенная поддержка глобального объекта `JSON`, возникла необходимость отличать объект `JSON`, предоставляемый окружением JavaScript, от аналогичных объектов, создаваемых с помощью библиотек. Используя тот же прием, который был продемонстрирован в функции `isArray()`, многие разработчики стали создавать примерно такие функции:

```
function supportsNativeJSON() {  
    return typeof JSON !== "undefined" &&  
        Object.prototype.toString.call(JSON) === "[object JSON]";  
}
```

Та же особенность метода `Object.prototype.toString()`, которая позволила разработчикам идентифицировать массивы, передаваемые через границы фреймов, дала возможность отличать

встроенные объекты JSON. Для объекта JSON, реализованного в библиотеке, возвращалась строка "[object Object]", а для встроенной версии — строка "[object JSON]". Такой подход стал фактическим стандартом для идентификации встроенных объектов.

Определение строковых меток для объектов в ECMAScript 6

Спецификация ECMAScript 6 сломала тенденцию встроенных объектов сообщать о своей идентичности через `Object.prototype.toString()`, добавив символ `Symbol.toStringTag`. Этот символ представляет свойство, имеющееся в любом объекте, которое определяет, какое значение должен возвращать вызов `Object.prototype.toString.call()` для данного объекта. Для массивов, например, возвращаемое этой функцией значение определяется строкой "Array", хранящейся в свойстве `Symbol.toStringTag`.

Имеется также возможность изменять значение `Symbol.toStringTag` в своих объектах:

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Person";

var me = new Person("Nicholas");

console.log(me.toString());           // "[object Person]"
console.log(Object.prototype.toString.call(me)); // "[object Person]"
```

Здесь определяется значение свойства `Symbol.toStringTag` в `Person.prototype`, чтобы обеспечить создание строкового представления по умолчанию. Так как `Person.prototype` наследует метод `Object.prototype.toString()`, значение свойства `Symbol.toStringTag` также возвращается вызовом метода `me.toString()`. Однако вы все еще можете определить свою версию метода `toString()` и реализовать иное поведение, не оказывая влияния на поведение метода `Object.prototype.toString.call()`. Вот как могла бы выглядеть такая реализация:

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Person";

Person.prototype.toString = function() {
    return this.name;
};

var me = new Person("Nicholas");

console.log(me.toString());           // "Nicholas"
console.log(Object.prototype.toString.call(me)); // "[object Person]"
```

В этом примере определяется метод `Person.prototype.toString()`, возвращающий значение свойства `name`. Поскольку экземпляры `Person` больше не наследуют метод `Object.prototype.toString()`, вызов `me.toString()` демонстрирует изменившееся поведение.

ПРИМЕЧАНИЕ

Все объекты наследуют свойство `Symbol.toStringTag` от `Object.prototype`, если оно не было переопределено явно. По умолчанию это свойство хранит строку `"Object"`.

Нет никаких ограничений на значения, которые можно хранить в свойствах `Symbol.toStringTag` объектов, созданных разработчиком. Например, ничто не мешает вам присвоить строку `"Array"` свойству `Symbol.toStringTag` своего объекта, например:

```
function Person(name) {
    this.name = name;
}

Person.prototype[Symbol.toStringTag] = "Array";

Person.prototype.toString = function() {
    return this.name;
};

var me = new Person("Nicholas");

console.log(me.toString());           // "Nicholas"
console.log(Object.prototype.toString.call(me)); // "[object Array]"
```

В этом примере вызов `Object.prototype.toString()` вернул строку `"[object Array]"`, которая также возвращается и для настоящих массивов. Этот пример лишний раз подчеркивает, что метод `Object.prototype.toString()` больше не может считаться надежным средством идентификации типов объектов.

Для встроенных объектов также можно изменить строковую метку. Достаточно присвоить новое значение свойству `Symbol.toStringTag` прототипа объекта, например:

```
Array.prototype[Symbol.toStringTag] = "Magic";

var values = [];

console.log(Object.prototype.toString.call(values)); // "[object Magic]"
```

В этом примере изменяется значение свойства `Symbol.toStringTag` для массивов, поэтому вызов `Object.prototype.toString()` вернул строку `"[object Magic]"` вместо `"[object Array]"`. Хотя поступать так не рекомендуется, в языке нет ничего, что запрещало бы подобные действия в отношении встроенных объектов.

Свойство `Symbol.unscopables`

Инструкция `with` является одной из конструкций языка JavaScript, вызывающих самые жаркие споры. Первоначально задумывавшаяся как средство, помогающее избежать ввода избыточного программного кода, инструкция `with` подвергается жесткой критике за то, что делает код трудночитаемым, отрицательно сказывается на производительности и способствует появлению ошибок. В результате инструкция `with` была сделана недопустимой в строгом режиме; это ограничение затрагивает также классы и модули, которые по умолчанию включают строгий режим, не давая никакой возможности обойти это условие.

Несмотря на то что в будущем инструкция `with`, вне всяких сомнений, будет исключена из языка, ECMAScript 6 все еще поддерживает ее в нестрогом режиме для обратной совместимости, то есть с целью не нарушить работу существующего кода, использующего `with`.

Чтобы понять суть проблемы, рассмотрим следующий фрагмент:

```
var values = [1, 2, 3],
    colors = ["red", "green", "blue"],
    color = "black";

with(colors) {
    push(color);
    push(...values);
}

console.log(colors);    // ["red", "green", "blue", "black", 1, 2, 3]
```

В этом примере два вызова `push()` внутри инструкции `with` эквивалентны вызовам `colors.push()`, потому что `with` добавляет `push` как локальную привязку. Ссылка `color` указывает на переменную, созданную за пределами инструкции `with`, то же относится и к ссылке `values`.

Но ECMAScript 6 добавила в массивы новый метод `values`. (Метод `values()` подробно обсуждается в главе 8.) В результате в окружении ECMAScript 6 ссылка `values` внутри инструкции `with` должна ссылаться не на локальную переменную `values`, а на метод `values` массивов, что может нарушить нормальную работу этого кода. Именно по этой причине был добавлен символ `Symbol.unscopables`.

Символ `Symbol.unscopables` используется в `Array.prototype` с целью показать, для каких свойств инструкция `with` не должна создавать локальные привязки. Когда свойство `Symbol.unscopables` определено и является объектом, его ключи со значением `true` интерпретируются как идентификаторы свойств родительского объекта, для которых инструкция `with` не должна создавать локальные привязки. Ниже приводится содержимое по умолчанию свойства `Symbol.unscopables` для массивов:

```
// встроено в ECMAScript 6 по умолчанию
Array.prototype[Symbol.unscopables] = Object.assign(Object.create(null), {
    copyWithin: true,
    entries: true,
    fill: true,
    find: true,
    findIndex: true,
    keys: true,
    values: true
});
```

Объект `Symbol.unscopables` имеет значение `null` в своем свойстве `prototype`, что получается в результате создания объекта вызовом `Object.create(null)`, и содержит все новые методы массивов, появившиеся в ECMAScript 6. (Эти методы подробно описываются в главах 8 и 10.) Инструкция `with` не создает локальные привязки для этих методов, благодаря чему старый код сохраняет работоспособность.

В общем случае нет необходимости определять свойство `Symbol.unscopables` для своих объектов, если они не используются в инструкциях `with`, или вносить изменения в существующие объекты.

В заключение

Символы — это новый тип элементарных значений в JavaScript, который используется для создания перечислимых свойств, недоступных без ссылки на символ. Такие свойства не являются по-настоящему приватными, тем не менее изменить или затереть их по ошибке существенно сложнее, а значит, они позволяют повысить уровень безопасности для функциональных возможностей, требующих дополнительной защиты от разработчиков.

Символы можно снабжать описаниями, упрощающими их идентификацию. Поддержка глобального реестра символов дает возможность использовать в разных фрагментах кода общие символы. Таким образом, один и тот же символ можно использовать в разных местах.

Методы, такие как `Object.keys()` или `Object.getOwnPropertyNames()`, не возвращают символы, поэтому спецификация ECMAScript 6 добавила новый метод `Object.getOwnPropertySymbols()`, чтобы дать возможность извлекать символьные свойства. Для определения символьных свойств по-прежнему можно использовать методы `Object.defineProperty()` и `Object.defineProperties()`.

Стандартные символы открывают доступ к внутренней функциональности стандартных объектов и определяются как глобальные константы-символы, такие как свойство `Symbol.hasInstance`. В спецификации эти символы начинаются с префикса `Symbol.` и позволяют разработчикам изменять поведение стандартных объектов.

Множества и ассоциативные массивы

Долгое время в JavaScript имелся только один тип коллекций — тип `Array`. (Некоторые разработчики утверждают, что все объекты помимо массивов являются коллекциями пар ключ/значение, однако следует заметить, что первоначально они имели иное предназначение, отличное от массивов.) Массивы в JavaScript используются точно так же, как массивы в других языках, но до появления ECMAScript 6 из-за нехватки коллекций других видов массивы часто применялись как очереди и стеки. Массивы позволяют использовать только числовые индексы, поэтому, когда возникает потребность в нечисловых индексах, разработчики используют объекты, не являющиеся массивами. Это привело к созданию нестандартных реализаций множеств и ассоциативных массивов на основе простых объектов.

Множество (set) — это список неповторяющихся значений. Обычно при использовании множества нет необходимости обращаться к отдельным его элементам, как в случае с массивами; гораздо чаще требуется просто проверить присутствие в множестве некоторого значения. *Ассоциативный массив (map)* — это коллекция ключей, соответствующих определенным значениям. Каждый элемент в ассоциативном массиве хранит два фрагмента данных, и значение извлекается из такого массива по указанному ключу. Ассоциативные массивы часто применяются для кэширования данных, которыми часто будут пользоваться в программе позднее. Спецификация ECMAScript 5 не определяет множества и ассоциативные массивы, тем не менее разработчики обходят это ограничение с применением обычных объектов.

ECMAScript 6 добавила множества и ассоциативные массивы в JavaScript, и эта глава обсуждает все, что вам следует знать об этих двух типах коллекций. Сначала я расскажу об обходных решениях, использовавшихся разработчиками для реализации множеств и ассоциативных массивов до появления ECMAScript 6, и связанных с ними проблемах. Затем опишу, как действуют множества и ассоциативные массивы в ECMAScript 6.

Множества и ассоциативные массивы в ECMAScript 5

В ECMAScript 5 разработчики имитировали множества и ассоциативные массивы, используя свойства объектов, как показано ниже:

```
var set = Object.create(null);

set.foo = true;

// проверка наличия
if (set.foo) {
```

```
// код для выполнения  
}
```

Переменная `set` в данном примере — это объект со значением `null` в свойстве `prototype`, гарантирующим отсутствие в объекте любых унаследованных свойств. Прием применения для проверки уникальных свойств объектов широко использовался в ECMAScript 5. Когда свойство добавляется в объект, ему присваивается значение `true`, чтобы условные инструкции (такие, как `if` в данном примере) легко могли проверить присутствие значения.

Единственное существенное отличие объекта, используемого в качестве множества, от объекта, используемого в качестве ассоциативного массива, заключается в сохраняемом значении. Например, в следующем примере объект используется как ассоциативный массив:

```
var map = Object.create(null);  
map.foo = "bar";  
  
// извлечь значение  
var value = map.foo;  
  
console.log(value);    // "bar"
```

Этот фрагмент сохраняет строковое значение `"bar"` с ключом `foo`. В отличие от множеств, ассоциативные массивы в основном используются для извлечения информации, а не для проверки существования ключа.

Недостатки обходных решений

Обычные объекты неплохо справляются с ролью множеств и ассоциативных массивов в простых ситуациях, но такой подход может усложнить дело из-за ограниченной природы свойств объектов. Например, поскольку имена всех свойств объектов должны быть строками, вам придется убедиться, что никакие два ключа не могут быть представлены в виде одной и той же строки. Взгляните на следующий пример:

```
var map = Object.create(null);  
map[5] = "foo";  
  
console.log(map["5"]);    // "foo"
```

В этом примере строковое значение `"foo"` присваивается числовому ключу `5`. Внутренне это числовое значение преобразуется в строку, поэтому `map["5"]` и `map[5]` в действительности ссылаются на одно и то же свойство. Это внутреннее преобразование может вызывать проблемы, когда приложению требуется использовать числовые и строковые ключи. Другая проблема возникает, когда в качестве ключей применяются объекты, как в следующем примере:

```
var map = Object.create(null),  
    key1 = {},  
    key2 = {};  
map[key1] = "foo";  
  
console.log(map[key2]);    // "foo"
```

Здесь `map[key2]` и `map[key1]` ссылаются на одно и то же значение. Объекты `key1` и `key2` преобразуются в строки, потому что имена свойств объектов должны быть строками. Так как по умолчанию объекты имеют строковое представление `"[object Object]"`, оба объекта, `key1` и

`key2`, преобразуются в эту строку. Это может вызывать трудноуловимые ошибки, потому что логично предположить, что разные ключи-объекты действительно должны быть разными.

Преобразование в строковое значение по умолчанию усложняет использование объектов в качестве ключей.

Отдельная проблема — ассоциативные массивы с ключами, представленными ложными значениями. В контекстах, где требуется логическое значение (например, в условном выражении в инструкции `if`), ложные значения автоматически преобразуются в `false`. Это преобразование не является проблемой, пока вы с особым вниманием относитесь к использованию значений. Например, взгляните на следующий фрагмент:

```
var map = Object.create(null);

map.count = 1;

// Что здесь проверяется? Присутствие ключа "count"
// или неравенство значения нулю?
if (map.count) {
    // код для выполнения
}
```

Этот пример допускает неоднозначную трактовку использования `map.count`. Что должна проверить инструкция `if`? Присутствие ключа `map.count` или неравенство его значения нулю? В данном случае код внутри инструкции `if` будет выполнен, потому что значение `1` — истинное. Однако если свойство `map.count` имеет значение `0` или вообще отсутствует, код внутри инструкции `if` выполнен не будет.

Эти проблемы, сложные в диагностике и отладке, когда возникают в крупных приложениях, стали основной причиной, почему спецификация ECMAScript 6 добавила в язык множества и ассоциативные массивы.

ПРИМЕЧАНИЕ

В JavaScript имеется оператор `in`, возвращающий `true`, если свойство присутствует в объекте, игнорируя его значение. Однако оператор `in` выполняет поиск в прототипе объекта, что делает его безопасным, только когда объект создается с пустым прототипом. Но, несмотря на это, многие разработчики все еще ошибочно используют код, как в предыдущем примере, вместо применения оператора `in`.

Множества в ECMAScript 6

Спецификация ECMAScript 6 добавляет тип `Set` — упорядоченный¹ список неповторяющихся значений. Множества дают быстрый доступ к содержащимся в них данным, поддерживая более эффективный способ извлечения дискретных значений.

Создание множеств и добавление элементов

Множества создаются с помощью `new Set()`, а добавление элементов производится вызовом метода `add()`. С помощью свойства `size` можно узнать количество элементов в множестве:

```
let set = new Set();
set.add(5);
```

¹ Под упорядоченностью здесь подразумевается «в порядке добавления» и ни в каком другом порядке. — *Примеч. пер.*

```
set.add("5");

console.log(set.size);    // 2
```

Множества не выполняют преобразование значений, проверяя их равенство. Это означает, что множество может содержать число 5 и строку "5" в двух разных элементах. (Внутренне сравнение значений выполняется с помощью метода `Object.is()`, обсуждавшегося в главе 4.) Вы можете также добавить в множество несколько объектов, и эти объекты будут считаться отличающимися:

```
let set = new Set(),
    key1 = {},
    key2 = {};

set.add(key1);
set.add(key2);

console.log(set.size);    // 2
```

Так как `key1` и `key2` не преобразуются в строки, они считаются двумя уникальными элементами в множестве. Если бы они преобразовывались в строки, оба получили бы одно и то же значение `"[object Object]"`.

Если вызвать метод `add()` несколько раз с одним и тем же значением, все вызовы, кроме первого, будут просто проигнорированы:

```
let set = new Set();
set.add(5);
set.add("5");
set.add(5);                // повторяющееся значение – игнорируется

console.log(set.size);    // 2
```

Здесь множество `set` имеет размер 2, потому что второе значение 5 не было добавлено в него. Также допускается инициализировать множества массивами, при этом конструктор `Set` оставит только уникальные значения. Например:

```
let set = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
console.log(set.size);    // 5
```

В этом примере для инициализации множества используется массив с повторяющимися значениями. Число 5 присутствует в множестве в единственном экземпляре, несмотря на то что в массиве это значение содержат четыре элемента. Такая возможность упрощает перевод существующего кода или структур JSON на использование множеств.

ПРИМЕЧАНИЕ

В действительности, конструктор `Set` принимает любые итерируемые объекты в качестве аргументов. Массивы поддерживаются по той простой причине, что они являются итерируемыми по умолчанию, так же как множества и ассоциативные массивы. Для извлечения значений из аргумента конструктор `Set` использует итератор. Итерируемые объекты и итераторы обсуждаются в главе 8.

Проверить присутствие значения в множестве можно с помощью метода `has()`, например:


```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true
console.log(set.has(6));    // false
```

Здесь `set.has(6)` возвращает `false`, потому что это значение отсутствует в множестве.

Удаление элементов

Существует также возможность удалять элементы из множеств. Вызов метода `delete()` удалит один элемент, а вызов метода `clear()` удалит все элементы. Следующий пример демонстрирует применение обоих методов:

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true

set.delete(5);

console.log(set.has(5));    // false
console.log(set.size);      // 1

set.clear();

console.log(set.has("5"));  // false
console.log(set.size);      // 0
```

Вызов `delete()` в этом примере удалил только элемент 5; вызов `clear()` очистил множество, удалив все элементы.

Множества — очень простой механизм поддержки уникальности упорядоченных значений. Но что если после заполнения множества потребуется выполнить некоторые операции с каждым элементом? Для этой цели существует метод `forEach()`.

Метод `forEach()` для множеств

Если вам приходилось плотно работать с массивами, возможно, вы уже знакомы с методом `forEach()`. Спецификация ECMAScript 5 добавила метод `forEach()` в массивы, чтобы упростить обработку каждого элемента массива без использования цикла `for`. Он завоевал большую популярность среди разработчиков, поэтому в множества был добавлен точно такой же метод.

Метод `forEach()` принимает функцию обратного вызова с тремя параметрами:

- Значение текущего элемента в множестве.
- То же значение, что и в первом аргументе.
- Ссылка на множество, которому принадлежит элемент.

Версия `forEach()` для множеств имеет одно странное отличие от версии для массивов — первый и второй аргументы функции обратного вызова в версии для множеств получают одно и то

же значение. На первый взгляд такое положение вещей выглядит как ошибка, но оно имеет вполне разумное объяснение.

Другие объекты, обладающие методом `forEach()` (простые и ассоциативные массивы), передают в свои функции обратного вызова три аргумента. В первых двух аргументах для простых и ассоциативных массивов передаются значение и ключ (числовой индекс для простых массивов).

Однако множества не имеют ключей. Люди, стоящие за стандартом ECMAScript 6, могли бы определить функцию обратного вызова для метода `forEach()` множеств как принимающую два аргумента, но тогда ее сигнатура получилась бы отличной от двух других версий. Вместо этого они сочли возможным сохранить сигнатуру функции обратного вызова как функции с тремя аргументами: каждое значение в множестве считается ключом и значением одновременно. В результате для сохранения функционального единообразия с методами `forEach()` простых и ассоциативных массивов первый и второй аргументы функции обратного вызова в версии метода `forEach()` множеств всегда получают одно и то же значение.

Кроме разницы в аргументах, метод `forEach()` множеств используется практически так же, как одноименный метод массивов. Следующий пример демонстрирует применение метода:

```
let set = new Set([1, 2]);

set.forEach(function(value, key, ownerSet) {
  console.log(key + " " + value);
  console.log(ownerSet === set);
});
```

Этот фрагмент выполняет итерации по всем элементам множества и выводит значения, передаваемые методом `forEach()` в функцию обратного вызова. При каждом вызове функции она получает аргументы `key` и `value` с одинаковыми значениями, а в аргументе `ownerSet` — ссылку на `set`. Этот фрагмент выведет:

```
1 1
true
2 2
true
```

По аналогии с массивами во втором аргументе методу `forEach()` можно передать ссылку `this`, если она потребуется в функции обратного вызова:

```
let set = new Set([1, 2]);

let processor = {
  output(value) {
    console.log(value);
  },
  process(dataSet) {
    dataSet.forEach(function(value) {
      this.output(value);
    }, this);
  }
};

processor.process(set);
```

В этом примере метод `processor.process()` вызывает `forEach()` множества `set` и передает ему ссылку `this` для использования в функции обратного вызова. Это необходимо, чтобы вызов

`this.output()` правильно интерпретировался как обращение к методу `processor.output()`. Здесь функция обратного вызова, что передается в вызов `forEach()`, использует только первый аргумент, поэтому остальные были опущены. Тот же самый эффект, но без передачи второго аргумента, можно получить с помощью стрелочной функции:

```
let set = new Set([1, 2]);

let processor = {
  output(value) {
    console.log(value);
  },
  process(dataSet) {
    dataSet.forEach(value => this.output(value));
  }
};

processor.process(set);
```

Стрелочная функция в этом примере получает значение `this` от вмещающей функции `process()`, поэтому она правильно интерпретирует вызов `this.output()` как `processor.output()`.

Имейте в виду, что хотя множества являются упорядоченным списком значений и позволяют последовательно обрабатывать элементы с помощью метода `forEach()`, вы не можете напрямую обращаться к элементам по индексам, как в случае с обычным массивом. Для этого лучше всего преобразовать множество в массив.

Преобразование множества в массив

Чтобы преобразовать массив в множество, достаточно просто передать этот массив в конструктор `Set`; обратное преобразование выполняется так же просто, если применить оператор расширения (`...`). В главе 3 оператор расширения был представлен как инструмент разбиения массивов на отдельные аргументы в вызовах функций. Однако этот оператор способен также преобразовывать итерируемые объекты, такие как множества, в массивы. Например:

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];

console.log(array);    // [1,2,3,4,5]
```

Здесь сначала в множество загружается массив, содержащий повторяющиеся элементы. Множество удаляет дубликаты, и затем с помощью оператора расширения его элементы помещаются в новый массив. Множество содержит элементы (1,2,3,4 и 5), полученные им в момент создания, и они просто копируются в новый массив.

Этот прием удобно использовать, когда имеется некоторый массив и требуется создать новый массив без повторяющихся значений, например:

```
function eliminateDuplicates(items) {
  return [...new Set(items)];
}

let numbers = [1, 2, 3, 3, 3, 4, 5],
    noDuplicates = eliminateDuplicates(numbers);

console.log(noDuplicates);    // [1,2,3,4,5]
```

Функция `eliminateDuplicates()` создает временное множество только с целью удалить повторяющиеся значения перед созданием нового массива.

Множества со слабыми ссылками

Тип Set можно назвать строгим множеством из-за особенностей хранения ссылок на объекты. Сохранение объекта в экземпляре Set фактически эквивалентно сохранению объекта в переменной. Пока ссылка на объект присутствует в таком экземпляре Set, объект не будет утилизирован сборщиком мусора. Например:

```
let set = new Set(),
    key = {};

set.add(key);
console.log(set.size);    // 1

// удалить оригинальную ссылку
key = null;

console.log(set.size);    // 1

// извлечь оригинальную ссылку обратно
key = [...set][0];
```

В этом примере присваивание значения `null` переменной `key` удаляет одну ссылку на объект `key`, но остается еще одна ссылка внутри `set`. Вы все еще можете извлечь `key`, преобразовав множество в массив с помощью оператора расширения и обратившись к первому элементу. Эта особенность отлично подходит для большинства программ, но иногда бывает желательно, чтобы ссылка исчезала из множества после удаления всех остальных ссылок. Например, если код на JavaScript выполняется в веб-странице и должен следить за элементами DOM, которые могут удаляться другим сценарием, было бы нежелательно, чтобы ваш код удерживал последнюю оставшуюся ссылку на элемент DOM. (Эта ситуация называется *утечкой памяти*.)

Для решения этой проблемы ECMAScript 6 включает также *множества со слабыми ссылками* (*weak sets*), которые хранят только слабые ссылки на объекты и не могут хранить элементарные значения. *Слабая ссылка* (*weak reference*) не препятствует утилизации объекта, если она — единственная оставшаяся ссылка.

Создание множеств со слабыми ссылками

Множества со слабыми ссылками создаются с помощью конструктора `WeakSet` и имеют методы `add()`, `has()` и `delete()`. Ниже приводится пример использования всех трех методов:

```
let set = new WeakSet(),
    key = {};

// добавить объект в множество
set.add(key);

console.log(set.has(key));    // true

set.delete(key);

console.log(set.has(key));    // false
```

Множества со слабыми ссылками используются практически так же, как обычные множества. Вы можете добавлять, удалять и проверять ссылки в таких множествах. Кроме того, множество со слабыми ссылками можно создать, передав в конструктор итерируемый объект:

```
let key1 = {},
    key2 = {},
    set = new WeakSet([key1, key2]);

console.log(set.has(key1));    // true
console.log(set.has(key2));    // true
```

В этом примере в вызов конструктора `WeakSet` передается массив. Так как этот массив содержит два объекта, эти объекты добавляются в множество со слабыми ссылками. Имейте в виду, что если в массиве окажется хотя бы одно значение, не являющееся объектом, это приведет к ошибке, потому что `WeakSet` не принимает элементарных значений.

Ключевые отличия между двумя типами множеств

Самое большое отличие множеств со слабыми ссылками от обычных множеств заключается в том, что они хранят слабые ссылки на объекты. Следующий пример демонстрирует это отличие:

```
let set = new WeakSet(),
    key = {};

// добавить объект в множество
set.add(key);

console.log(set.has(key));    // true

// удалить последнюю строгую ссылку на key
// (ссылка также будет удалена из множества)
key = null;
```

После того как этот код выполнится, ссылка на `key` в множестве со слабыми ссылками окажется недоступной. Проверить этот факт невозможно, потому что для этого потребовалось бы передать ссылку на объект в вызов метода `has()`. Это обстоятельство может превратить тестирование множеств со слабыми ссылками в запутанную задачу, но вы можете верить, что ссылка действительно удаляется движком JavaScript.

Как показывает предыдущий пример, множества со слабыми ссылками обладают почти теми же характеристиками, что и обычные множества, но существуют некоторые важные отличия:

- Методы `add()`, `has()` и `delete()` экземпляра `WeakSet` возбуждают ошибку при передаче им значения, не являющегося объектом.
- Множества со слабыми ссылками не являются итерируемыми объектами и потому не могут использоваться в цикле `for-of`.
- Множества со слабыми ссылками не экспортируют итераторов (таких, как методы `keys()` и `values()`), поэтому нет никакой возможности программно определить содержимое множества со слабыми ссылками.
- Множества со слабыми ссылками не имеют метода `forEach()`.
- Множества со слабыми ссылками не имеют свойства `size`.

Такие ограничения возможностей множеств со слабыми ссылками, по-видимому, необходимы для корректной работы с памятью. Вообще говоря, если вам требуется только следить за ссылками на объекты, используйте множества со слабыми ссылками вместо обычных множеств.

Множества дают новые способы обработки списков значений, но они не особенно полезны, когда требуется сопроводить эти значения дополнительной информацией. Именно поэтому в ECMAScript 6 были добавлены ассоциативные массивы.

Ассоциативные массивы в ECMAScript 6

Тип `Map` в ECMAScript 6 — это упорядоченный² список пар ключ/значение, где ключ и значение могут быть любого типа. Эквивалентность ключей определяется вызовом метода `Object.is()`, поэтому можно иметь ключи `5` и `"5"`, так как они имеют разные типы. Это существенно отличает их от свойств объектов, потому что имена свойств объектов всегда преобразуются в строки.

В ассоциативные массивы можно добавлять элементы вызовом метода `set()`, передавая ему ключ и значение, связанное с ключом. Извлечь значение можно вызовом метода `get()`, передав ему соответствующий ключ. Например:

```
let map = new Map();
map.set("title", "Understanding ECMAScript 6");
map.set("year", 2016);

console.log(map.get("title")); // "Understanding ECMAScript 6"
console.log(map.get("year")); // 2016
```

В этом примере сохраняются две пары ключ/значение. С ключом `"title"` сохраняется строка, а с ключом `"year"` — число. Затем вызывается метод `get()` для извлечения значений обоих ключей. Если ключ отсутствует в ассоциативном массиве, `get()` вернет специальное значение `undefined`.

В качестве ключей также допускается использовать объекты, что невозможно, когда ассоциативный массив имитируется с применением простых свойств объекта. Например:

```
let map = new Map(),
    key1 = {},
    key2 = {};

map.set(key1, 5);
map.set(key2, 42);

console.log(map.get(key1)); // 5
console.log(map.get(key2)); // 42
```

В этом примере в качестве ключей используются объекты `key1` и `key2`, для которых в ассоциативном массиве сохраняются два разных значения. Поскольку ключи не преобразуются ни в какую другую форму, каждый объект считается уникальным. Это позволяет ассоциировать с объектами дополнительную информацию без изменения самих объектов.

Методы ассоциативных массивов

Ассоциативные массивы имеют те же методы, что и множества. Так было задумано изначально, чтобы дать возможность взаимодействовать с ассоциативными массивами и множествами похожими способами. Следующие три метода доступны в ассоциативных массивах и множествах:

- `has(key)`. Определяет присутствие указанного ключа в ассоциативном массиве.
- `delete(key)`. Удаляет из ассоциативного массива указанный ключ и связанное с ним значение.
- `clear()`. Удаляет из ассоциативного массива все ключи вместе со значениями.

² Под упорядоченностью здесь подразумевается «в порядке добавления» и ни в каком другом порядке. — *Примеч. пер.*

Ассоциативные массивы имеют также свойство `size`, возвращающее количество пар ключ/значение, содержащихся в них. Следующий фрагмент демонстрирует использование всех трех методов и свойства `size`:

```
let map = new Map();
map.set("name", "Nicholas");
map.set("age", 25);

console.log(map.size);           // 2

console.log(map.has("name"));    // true
console.log(map.get("name"));    // "Nicholas"
console.log(map.has("age"));     // true
console.log(map.get("age"));     // 25

map.delete("name");
console.log(map.has("name"));    // false
console.log(map.get("name"));    // undefined
console.log(map.size);          // 1

map.clear();
console.log(map.has("name"));    // false
console.log(map.get("name"));    // undefined
console.log(map.has("age"));     // false
console.log(map.get("age"));     // undefined
console.log(map.size);          // 0
```

Так же как во множествах, свойство `size` всегда возвращает количество пар ключ/значение в ассоциативном массиве. Экземпляр `Map` в этом примере сначала получает ключи `"name"` и `"age"`, поэтому `has()` возвращает `true`, когда ему передается любой из этих ключей. После удаления ключа `"name"` вызовом метода `delete()` метод `has()` возвращает `false` для ключа `"name"`, а свойство `size` показывает, что количество элементов уменьшилось на единицу. Затем вызовом метода `clear()` удаляется оставшийся ключ, о чем сообщает метод `has()`, возвращающий `false` для обоих ключей, а свойство `size` получает значение 0.

Метод `clear()` позволяет быстро удалить большой объем данных из ассоциативного массива, но существует также способ сразу добавить большой объем данных.

Инициализация ассоциативных массивов

Ассоциативные массивы, как и множества, можно инициализировать передачей массива в конструктор `Map`. Каждый элемент такого массива сам должен быть массивом с двумя элементами, первый из которых определяет ключ, а второй — значение, соответствующее этому ключу. Следовательно, ассоциативный массив, по сути, является массивом двухэлементных массивов, например:

```
let map = new Map([["name", "Nicholas"], ["age", 25]]);

console.log(map.has("name"));    // true
console.log(map.get("name"));    // "Nicholas"
console.log(map.has("age"));     // true
console.log(map.get("age"));     // 25
console.log(map.size);          // 2
```

В этом примере посредством инициализации через конструктор в ассоциативный массив `map` добавляются ключи `"name"` и `"age"`. Несмотря на то что массив массивов выглядит немного странным, такая организация необходима для точного представления ключей, потому что ключи могут быть любого типа. Сохранение ключей в массиве — единственный способ избежать их преобразования в другой тип данных перед сохранением в ассоциативном массиве.

Метод `forEach()` ассоциативных массивов

Метод `forEach()` ассоциативных массивов напоминает метод `forEach()` для множеств и простых массивов — он принимает функцию обратного вызова с тремя параметрами:

- Значение текущего элемента в ассоциативном массиве.
- Ключ этого значения.
- Ссылка на ассоциативный массив, из которого извлечен элемент.

Аргументы функции обратного вызова больше соответствуют аргументам аналогичной функции, которая передается методу `forEach()` простых массивов: в первом передается значение, а во втором — ключ (соответствует числовому индексу в случае с простыми массивами). Например:

```
let map = new Map([["name", "Nicholas"], ["age", 25]]);

map.forEach(function(value, key, ownerMap) {
  console.log(key + " " + value);
  console.log(ownerMap === map);
});
```

Функция обратного вызова, которая передается методу `forEach()` в этом примере, выводит полученную ею информацию. Аргументы `value` и `key` выводятся непосредственно, а аргумент `ownerMap` сравнивается с `map`, чтобы показать, что эти две ссылки эквивалентны. Этот пример выведет следующее:

```
name Nicholas
true
age 25
true
```

Функция обратного вызова, которая передается в вызов `forEach()`, получает пары ключ/значение в порядке их добавления в ассоциативный массив. Это отличается от поведения метода `forEach()` простых массивов, который передает элементы функции обратного вызова в порядке возрастания числовых индексов.

ПРИМЕЧАНИЕ

Методу `forEach()` можно также передать второй аргумент — значение для ссылки `this` внутри функции обратного вызова. Такой вызов будет действовать в точности как версия метода `forEach()` для множеств.

Ассоциативные массивы со слабыми ссылками

Ассоциативные массивы со слабыми ссылками соответствуют ассоциативным массивам, как множества со слабыми ссылками соответствуют обычным множествам: они хранят слабые ссылки на объекты. В *ассоциативном массиве со слабыми ссылками (weak maps)* каждый ключ должен быть объектом (попытка использовать в качестве ключа элементарное значение вызовет ошибку), и ссылки на эти объекты являются слабыми, поэтому они не мешают утилизировать объекты. Когда за пределами ассоциативного массива со слабыми ссылками не останется ни одной ссылки на объект, используемый в качестве ключа, соответствующая пара ключ/значение будет автоматически удалена. Но слабые ссылки хранятся только в ключах, значения не могут хранить слабых ссылок. Ссылка на объект, хранящаяся в ассоциативном массиве со слабыми ссылками в виде значения, будет препятствовать утилизации объекта, даже если все остальные ссылки на этот объект будут удалены.

Ассоциативные массивы со слабыми ссылками могут очень пригодиться для хранения объектов, связанных с элементами DOM веб-страницы. Например, некоторые библиотеки на JavaScript для веб-страниц поддерживают собственные объекты для ссылки на элементы DOM внутри библиотеки, и такое отображение хранится во внутреннем кэше объектов.

Самое сложное в таком решении — определить момент, когда элемент DOM удаляется из веб-страницы, чтобы удалить соответствующий ему объект из кэша. Если этого не сделать, библиотека будет удерживать бесполезную ссылку на элемент DOM, что вызовет утечку памяти. Кэширование элементов DOM с применением ассоциативного массива со слабыми ссылками все же позволяет библиотеке связать каждый элемент DOM со своими объектами и автоматически удалять любые объекты из ассоциативного массива одновременно с удалением элемента DOM.

Использование ассоциативных массивов со слабыми ссылками

Тип `WeakMap`, появившийся в ECMAScript 6, — это неупорядоченный список пар ключ/значение, ключи которого должны быть действительными ссылками на объекты, а значения могут быть любого типа. Интерфейс `WeakMap` очень похож на интерфейс `Map` — для добавления и извлечения данных используются методы `set()` и `get()` соответственно:

```
let map = new WeakMap(),
    element = document.querySelector(".element");
```

```
map.set(element, "Original");
```

```
let value = map.get(element);
console.log(value);    // "Original"
```

```
// удалить элемент element
element.parentNode.removeChild(element);
element = null;
```

```
// в этой точке ассоциативный массив со слабыми ссылками оказывается пустым
```

В этом примере сохраняется одна пара ключ/значение. Ключ `element`, элемент DOM, используется для сохранения соответствующего строкового значения. Это значение затем извлекается передачей элемента DOM в вызов метода `get()`. Когда позднее элемент DOM удаляется из документа и переменной, ссылавшейся на него, присваивается значение `null`, происходит автоматическое удаление данных из ассоциативного массива со слабыми ссылками.

Как и в случае с множествами со слабыми ссылками, нет никакой возможности проверить, что ассоциативный массив со слабыми ссылками опустел, потому что он не имеет свойства `size`. А так как не осталось ссылки, что хранилась в ключе, вы не сможете извлечь значение вызовом метода

`get()`. Ассоциативный массив со слабыми ссылками закрывает доступ к значению для этого ключа, и когда сборщик мусора запустится, он освободит память, занятую значением.

Инициализация ассоциативных массивов со слабыми ссылками

Чтобы инициализировать ассоциативный массив со слабыми ссылками, передайте массив массивов в вызов конструктора `WeakMap`. Так же как в случае с обычными ассоциативными массивами, каждый вложенный массив должен содержать два элемента: первый элемент, если это непустая ссылка на объект, становится ключом, а второй (с данными любого типа) — значением. Например:

```
let key1 = {},
    key2 = {},
    map = new WeakMap([[key1, "Hello"], [key2, 42]]);

console.log(map.has(key1)); // true
console.log(map.get(key1)); // "Hello"
console.log(map.has(key2)); // true
console.log(map.get(key2)); // 42
```

Объекты `key1` и `key2` в этом примере становятся ключами в ассоциативном массиве со слабыми ссылками, и методы `get()` и `has()` позволяют обращаться к ним. Попытка передать конструктору `WeakMap` массив, в котором какая-либо пара ключ/значение имеет ключ, не являющийся объектом, вызовет ошибку.

Методы ассоциативных массивов со слабыми ссылками

Ассоциативные массивы со слабыми ссылками имеют только два дополнительных метода для работы с парами ключ/значение. Метод `has()` определяет присутствие ключа в ассоциативном массиве, а метод `delete()` удаляет указанную пару ключ/значение. Метод `clear()` отсутствует, потому что он мог бы потребовать возможности перечисления ключей, но в ассоциативных массивах со слабыми ссылками это невозможно, так же как и во множествах со слабыми ссылками. Следующий пример демонстрирует применение методов `has()` и `delete()`:

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

console.log(map.has(element)); // true
console.log(map.get(element)); // "Original"

map.delete(element);
console.log(map.has(element)); // false
console.log(map.get(element)); // undefined
```

Здесь снова в качестве ключа используется элемент DOM. Метод `has()` позволяет проверить присутствие в данный момент искомого ключа в ассоциативном массиве со слабыми ссылками. Имейте в виду, что этот прием дает положительный результат, только когда ключ хранит непустую ссылку. Метод `delete()` принудительно удаляет ключ из ассоциативного массива, после чего вызов `has()` возвращает `false`, а `get()` возвращает `undefined`.

Приватные данные объекта

Большинство разработчиков полагает, что основное назначение ассоциативных массивов со слабыми ссылками — хранение данных, ассоциированных с элементами DOM, однако существует много других областей применения (и, вне всяких сомнений, какие-то области остаются пока неоткрытыми). Например, ассоциативные массивы со слабыми ссылками можно использовать для хранения данных, приватных для экземпляров объектов. Все свойства объектов в ECMAScript 6 являются общедоступными, поэтому приходится проявлять смекалку, чтобы сделать данные доступными только внутри объекта. Взгляните на следующий пример:

```
function Person(name) {
    this._name = name;
}

Person.prototype.getName = function() {
    return this._name;
};
```

Здесь используется распространенное соглашение об использовании символа подчеркивания в качестве признака приватного свойства, значение которого не должно изменяться за пределами экземпляра объекта. Цель состоит в том, чтобы использовать `getName()` для чтения `this._name` и не позволить изменять `_name`. Однако ничто не мешает внешнему коду просто взять и записать в свойство `_name` новое значение — преднамеренно или по ошибке.

В ECMAScript 5 имеется возможность закрывать доступ к приватным данным с использованием следующего шаблонного решения:

```
var Person = (function() {

    var privateData = {},
        privateId = 0;

    function Person(name) {
        Object.defineProperty(this, "_id", { value: privateId++ });

        privateData[this._id] = {
            name: name
        };
    }

    Person.prototype.getName = function() {
        return privateData[this._id].name;
    };

    return Person;
})();
```

В этом примере определение `Person` завернуто в выражение немедленно вызываемой функции (Immediately Invoked Function Expression, IIFE). Объект содержит две приватные переменные, `privateData` и `privateId`. Объект в поле `privateData` хранит приватную информацию для каждого экземпляра, а `privateId` получает уникальный числовой идентификатор экземпляра. Когда вызывается конструктор `Person`, он добавляет неперечислимое, ненастраиваемое и недоступное для записи свойство `_id`.

Затем в объект `privateData` добавляется запись (еще один объект), соответствующая числовому идентификатору экземпляра; в этой записи сохраняется значение аргумента `name`. Позднее это

значение можно получить вызовом функции `getName()`, которая использует `this._id` в качестве ключа в `privateData`. Так как поле `privateData` недоступно за пределами области видимости функции IIFE, данные оказываются в безопасности, даже если значение `this._id` остается общедоступным.

Самая главная проблема такого решения в том, что данные в `privateData` никогда не исчезают, потому что нет никакой возможности определить момент удаления экземпляра объекта: объект `privateData` всегда хранит лишние данные. Эту проблему можно решить с помощью ассоциативного массива со слабыми ссылками, как показано ниже:

```
let Person = (function() {

    let privateData = new WeakMap();

    function Person(name) {
        privateData.set(this, { name: name });
    }

    Person.prototype.getName = function() {
        return privateData.get(this).name;
    };

    return Person;
})();
```

В этой версии `Person` для хранения приватных данных вместо объекта используется ассоциативный массив со слабыми ссылками. Так как в качестве ключей можно использовать сами экземпляры объекта `Person`, отпала необходимость в отдельном числовом идентификаторе. Когда вызывается конструктор `Person`, в ассоциативный массив со слабыми ссылками добавляется новый элемент с ключом `this` и объектом с приватной информацией в качестве значения. В данном случае значение — это объект с единственным свойством `name`. Функция `getName()` извлекает приватную информацию, передавая `this` в вызов метода `privateData.get()`, который извлекает объект-значение и возвращает его свойство `name`. Этот прием помогает сохранить приватную информацию недоступной и уничтожает эту информацию, как только будет уничтожен соответствующий ей экземпляр объекта.

Применение и ограничения ассоциативных массивов со слабыми ссылками

Когда наступит момент выбирать между простым ассоциативным массивом и ассоциативным массивом со слабыми ссылками, в первую очередь определите, собираетесь ли вы использовать в качестве ключей только объекты. Всякий раз когда для ключей предполагается использовать только объекты, ассоциативные массивы со слабыми ссылками оказываются лучшим выбором. Такие ассоциативные массивы позволяют оптимизировать использование памяти и избежать утечек, гарантируя автоматическое удаление лишних данных после того, как они станут недоступны.

Имейте в виду, что ассоциативные массивы со слабыми ссылками ограничивают доступ к содержащимся в них данным: вы не сможете использовать метод `forEach()`, свойство `size` или метод `clear()` для управления элементами. Если вам необходимы средства исследования элементов, лучше выбрать обычный ассоциативный массив. Но тогда просто помните о необходимости управления памятью.

Наконец, если в качестве ключей предполагается использовать элементарные значения, тогда обычные ассоциативные массивы — единственно возможный вариант.

В заключение

ECMAScript 6 официально ввела множества и ассоциативные массивы в JavaScript. Прежде для имитации множеств и ассоциативных массивов разработчики использовали обычные объекты, часто сталкиваясь с проблемами из-за ограничений, характерных для свойств объектов.

Множества — это упорядоченные списки уникальных значений. Значения считаются уникальными, если они не эквивалентны с точки зрения метода `Object.is()`. Множества автоматически удаляют повторяющиеся значения, поэтому множество можно использовать как фильтр для устранения повторяющихся элементов из массивов. Множества не являются подклассом массивов, поэтому они не поддерживают произвольный доступ к своим элементам. Вместо этого вам придется использовать метод `has()`, чтобы определить, присутствует ли значение в множестве, и свойство `size`, чтобы узнать количество значений в множестве. Тип `Set` имеет также метод `forEach()` для обработки всех значений в множестве.

Множества со слабыми ссылками — это особая разновидность множеств, которая позволяет хранить только объекты. Объекты сохраняются в виде слабых ссылок на них, то есть элемент со слабой ссылкой не препятствует утилизации объекта сборщиком мусора, если этот элемент оказался последней ссылкой на этот объект. Содержимое множества со слабыми ссылками нельзя исследовать из-за сложностей, связанных с управлением памятью, поэтому такие множества лучше использовать только для слежения за объектами.

Ассоциативные массивы — это упорядоченные списки пар ключ/значение, где ключом могут быть данные любого типа. По аналогии с множествами уникальность ключей определяется с помощью метода `Object.is()`, то есть числовой ключ `5` и строковый ключ `"5"` считаются двумя разными ключами. С ключом можно связать значение любого типа, вызвав метод `set()`. Позднее это значение можно извлечь с помощью метода `get()`. Ассоциативные массивы также имеют свойство `size` и метод `forEach()`, упрощающий обход всех элементов.

Ассоциативные массивы со слабыми ссылками — это особая разновидность ассоциативных массивов, которая позволяет использовать в качестве ключей только объекты. Так же как в множествах со слабыми ссылками, в ключах сохраняются слабые ссылки на объекты, не препятствующие их утилизации, когда слабая ссылка осталась единственной, указывающей на объект. Когда объект в ключе утилизируется сборщиком мусора, значение, связанное с ключом, также удаляется из ассоциативного массива со слабыми ссылками. Этот аспект управления памятью делает ассоциативные массивы со слабыми ссылками уникальным инструментом для связывания дополнительной информации с объектами, чей жизненный цикл управляется за пределами кода, обращающегося к ним.

Итераторы и генераторы

Многие языки программирования переходят от использования для итераций по данным циклов `for`, требующих инициализации переменных, которые определяют позицию в коллекции, к применению объектов-итераторов, возвращающих следующий элемент коллекции. Итераторы упрощают работу с коллекциями данных, поэтому ECM AScript 6 также добавила итераторы в JavaScript. В комплексе с новыми методами массивов и новыми типами коллекций (такими, как множества и ассоциативные массивы) итераторы оказываются эффективными инструментами обработки данных, и вы часто будете встречаться с ними в разных разделах языка. Новый цикл `for-of` работает с итераторами; оператор расширения (`...`) использует итераторы; и даже в асинхронном программировании можно применять итераторы.

Эта глава охватывает множество приемов применения итераторов, но прежде чем перейти к практическим примерам, необходимо ознакомиться с причинами, объясняющими включение итераторов в JavaScript.

Проблемы использования циклов

Если вам доводилось программировать на JavaScript, вы наверняка писали примерно такой код:

```
var colors = ["red", "green", "blue"];

for (var i = 0, len = colors.length; i < len; i++) {
    console.log(colors[i]);
}
```

Это стандартный способ использования цикла `for` для перебора индексов в массиве `colors` с применением переменной `i`. Значение `i` увеличивается на единицу в каждой итерации цикла, которые продолжаются, пока значение `i` не превысит значение длины массива (хранится в переменной `len`).

Цикл в этом примере выглядит очень просто, но сложность циклов увеличивается, когда приходится вкладывать их друг в друга и следить за множеством переменных. С ростом сложности увеличивается вероятность появления ошибок, и сама шаблонная природа цикла `for` способствует появлению ошибок, потому что в разных местах используется однотипный код. Итераторы помогают избавиться от лишних сложностей и обуздать природу циклов, способствующую появлению ошибок.

Что такое итераторы?

Итераторы — это объекты со специализированным интерфейсом, спроектированным для итераций. Все объекты-итераторы имеют метод `next()`, возвращающий объект результата. Объект результата имеет два свойства: `value` — следующее значение и `done`, принимающее значение `true`, когда в коллекции не остается элементов для последующих итераций. Итератор хранит внутренний указатель на местоположение значений в коллекции, и каждый вызов его метода `next()` возвращает следующее значение.

Если вызвать `next()` после извлечения последнего значения, метод вернет значение `true` в свойстве `done` объекта результата и *возвращаемое значение* итератора в свойстве `value`. Это возвращаемое значение не является частью коллекции данных; скорее, это заключительный фрагмент данных, связанных с коллекцией, или `undefined`, если такого фрагмента данных не существует. Возвращаемое значение итератора напоминает возвращаемое значение функции тем, что это последний способ передать информацию вызывающему коду.

С учетом вышеизложенного мы могли бы создать итератор в ECMAScript 5, как показано ниже:

```
function createIterator(items) {

    var i = 0;

    return {
        next: function() {

            var done = (i >= items.length);
            var value = !done ? items[i++] : undefined;

            return {
                done: done,
                value: value
            };
        }
    };
}

var iterator = createIterator([1, 2, 3]);

console.log(iterator.next());    // "{ value: 1, done: false }"
console.log(iterator.next());    // "{ value: 2, done: false }"
console.log(iterator.next());    // "{ value: 3, done: false }"
console.log(iterator.next());    // "{ value: undefined, done: true }"

// любой последующий вызов
console.log(iterator.next());    // "{ value: undefined, done: true }"
```

Функция `createIterator()` возвращает объект с методом `next()`. Каждый вызов этого метода возвращает следующее значение из массива `items` в свойстве `value`. Когда `i` достигает 3, свойство `done` получает значение `true`, а тернарный условный оператор записывает в свойство `value` значение `undefined`. Эти два результата соответствуют последнему специальному случаю итераторов в ECMAScript 6, когда метод `next()` итератора вызывается после достижения конца коллекции.

Как показывает этот пример, реализация итераторов в соответствии с правилами, изложенными в ECMAScript 6, — не самое простое занятие. К счастью, ECMAScript 6 поддерживает также генераторы, которые существенно упрощают реализацию итераторов.

Что такое генераторы?

Генератор — это функция, возвращающая итератор. Функции-генераторы отмечаются звездочкой (*) после ключевого слова `function` и используют новое ключевое слово `yield`. Совершенно неважно, следует ли звездочка сразу за ключевым словом `function` или отделяется от него пробельными символами, как в следующем примере:

```
// генератор
function *createIterator() {
    yield 1;
    yield 2;
    yield 3;
}

// генераторы вызываются как обычные функции,
// но возвращают итераторы
let iterator = createIterator();

console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
```

Символ `*` перед `createIterator()` превращает эту функцию в генератор. Ключевое слово `yield`, также впервые появившееся в ECMAScript 6, определяет значения, которые итератор должен возвращать при вызове его метода `next()`, и порядок возврата этих значений. Итератор, сгенерированный в этом примере, возвращает три разных значения в трех последовательных вызовах метода `next()`: сначала 1, затем 2 и, наконец, 3. Генератор вызывается как любая другая функция, в чем можно убедиться, взглянув на инструкцию, создающую `iterator`.

Самое интересное в функциях-генераторах заключается в том, что они останавливают выполнение после каждой инструкции `yield`. Например, после выполнения инструкции `yield1` в этом примере функция остановит выполнение до следующего вызова метода `next()` итератора. После чего она выполнит `yield2`. Такая способность приостанавливать выполнение в середине функции открывает некоторые интересные способы применения функций-генераторов (которые обсуждаются ниже в разделе «Дополнительные возможности итераторов»).

Ключевое слово `yield` может использоваться с любым значением или выражением, поэтому можно написать функцию-генератор, которая будет добавлять элементы в итераторы, а не просто перечислять их по одному. Например, ниже приводится один из вариантов использования `yield` в цикле `for`:

```
function *createIterator(items) {
    for (let i = 0; i < items.length; i++) {
        yield items[i];
    }
}

let iterator = createIterator([1, 2, 3]);

console.log(iterator.next()); // "{ value: 1, done: false }"
console.log(iterator.next()); // "{ value: 2, done: false }"
console.log(iterator.next()); // "{ value: 3, done: false }"
console.log(iterator.next()); // "{ value: undefined, done: true }"

// любой последующий вызов
console.log(iterator.next()); // "{ value: undefined, done: true }"
```


В этом примере функции-генератору `createIterator()` передается массив `items`. Внутри функции цикл `for` перебирает элементы массива и передает их в итератор. Каждый раз когда встречается инструкция `yield`, цикл останавливается до следующего вызова метода `next()` итератора, после чего цикл продолжает выполнение до следующей инструкции `yield`.

Функции-генераторы — важный элемент ECMAScript 6, а так как это всего лишь функции, вы можете использовать их везде, где можно использовать обычные функции. Остальная часть этого раздела посвящена другим интересным способам создания генераторов.

КОГДА YIELD ВОЗБУЖДАЕТ ОШИБКИ

Ключевое слово `yield` можно использовать только внутри генераторов. Попытка использовать его в любом другом месте, включая функции внутри генераторов, как в следующем примере, является синтаксической ошибкой:

```
function *createIterator(items) {  
    items.forEach(function(item) {  
        // синтаксическая ошибка  
        yield item + 1;  
    });  
}
```

Несмотря на то что технически `yield` находится внутри функции `createIterator()`, этот код вызовет синтаксическую ошибку, потому что `yield` не может пересекать границы функций. Ключевое слово `yield` напоминает `return` — вложенная функция не может вернуть значение своей вмещающей функции.

Выражения функций-генераторов

Для создания генераторов можно использовать функции-выражения. В этом случае звездочку (*) следует поместить между ключевым словом `function` и открывающей круглой скобкой. Например:

```
let createIterator = function *(items) {  
    for (let i = 0; i < items.length; i++) {  
        yield items[i];  
    }  
};
```

```
let iterator = createIterator([1, 2, 3]);
```

```
console.log(iterator.next()); // "{ value: 1, done: false }"  
console.log(iterator.next()); // "{ value: 2, done: false }"  
console.log(iterator.next()); // "{ value: 3, done: false }"  
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

```
// любой последующий вызов  
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

В этом примере `createIterator()` — это выражение функции-генератора. Звездочка находится между ключевым словом `function` и открывающей круглой скобкой, потому что выражения функций не имеют имен. В остальном этот пример ничем не отличается от предыдущего и также использует цикл `for`.

ПРИМЕЧАНИЕ

Создание стрелочных функций-генераторов невозможно.

Методы-генераторы объектов

Так как генераторы — это всего лишь функции, их можно добавлять в объекты. Например, ниже показано, как с помощью функции-выражения определить генератор в литерале объекта в стиле ECMAScript 5:

```
let o = {

  createIterator: function *(items) {
    for (let i = 0; i < items.length; i++) {
      yield items[i];
    }
  }
};

let iterator = o.createIterator([1, 2, 3]);
```

Можно также использовать сокращенный синтаксис определения методов в ECMAScript 6, поместив звездочку (*) перед именем метода, как показано ниже:

```
let o = {

  *createIterator(items) {
    for (let i = 0; i < items.length; i++) {
      yield items[i];
    }
  }
};

let iterator = o.createIterator([1, 2, 3]);
```

Эти примеры функционально эквивалентны примеру в предыдущем разделе; они отличаются только синтаксисом. В версии с использованием сокращенного синтаксиса, поскольку метод `createIterator()` определяется без ключевого слова `function`, звездочка помещается непосредственно перед именем метода, при этом допускается разделять звездочку и имя метода пробельными символами.

Итерируемые объекты и циклы `for-of`

С итераторами тесно связаны *итерируемые объекты* — объекты, обладающие свойством `Symbol.iterator`. Стандартный символ `Symbol.iterator` определяет функцию, возвращающую итератор для данного объекта. Все коллекции объектов (массивы, множества и ассоциативные массивы) и строки являются в ECMAScript 6 итерируемыми, поэтому для них определены итераторы по умолчанию. Итерируемые объекты создавались с целью использовать их в новом цикле, появившемся в ECMAScript, — в цикле `for-of`.

ПРИМЕЧАНИЕ

Все итераторы, созданные с помощью генераторов, сами являются итерируемыми объектами, потому что по умолчанию генераторы присваиваются свойству `Symbol.iterator`.

В начале этой главы я упоминал о проблеме, связанной с необходимостью поддерживать индекс внутри цикла `for`. Итераторы — первая часть решения этой проблемы. Вторая часть — цикл `for-of`: он полностью избавляет от необходимости поддерживать индекс в коллекции, позволяя сосредоточиться на содержимом коллекции.

В каждой итерации цикл `for-of` вызывает метод `next()` итерируемого объекта и сохраняет в переменной значение свойства `value` объекта результата. Итерации продолжаются, пока свойство `done` возвращаемого объекта не получит значение `true`. Например:

```
let values = [1, 2, 3];
```

```
for (let num of values) {  
  console.log(num);  
}
```

Этот пример выведет следующее:

```
1 2 3
```

Данный цикл `for-of` сначала вызовет метод `Symbol.iterator` массива `values`, чтобы получить итератор. (Вызов `Symbol.iterator` выполняется движком JavaScript неявно.) Затем вызовет `iterator.next()` и прочтает свойство `value` объекта результата в переменную `num`. Переменная `num` сначала получит значение 1, затем 2 и, наконец, 3. Когда свойство `done` объекта результата получит значение `true`, цикл завершится, поэтому переменная `num` никогда не получит значения `undefined`.

Если требуется просто обойти значения в массиве или коллекции, цикл `for-of` выглядит предпочтительнее, чем `for`. Вообще цикл `for-of` способствует уменьшению количества ошибок, потому что проверяется меньшее количество условий. Применяйте традиционный цикл `for`, только когда требуется реализовать более сложную логику управления.

ВНИМАНИЕ

Инструкция `for-of` возбуждает ошибку при попытке применить ее к неитерируемому объекту, значению `null` или `undefined`.

Доступ к итератору по умолчанию

Доступ к итератору по умолчанию объекта можно получить с помощью символа `Symbol.iterator`:

```
let values = [1, 2, 3];  
let iterator = values[Symbol.iterator]();  
  
console.log(iterator.next()); // "{ value: 1, done: false }"  
console.log(iterator.next()); // "{ value: 2, done: false }"  
console.log(iterator.next()); // "{ value: 3, done: false }"  
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

В этом примере извлекается итератор по умолчанию для массива `values`, который используется для итераций по элементам массива. То же самое происходит за кулисами цикла `for-of`.

Так как `Symbol.iterator` определяет итератор по умолчанию, с помощью этого свойства можно определить, является ли объект итерируемым, например:

```
function isIterable(object) {
    return typeof object[Symbol.iterator] === "function";
}

console.log(isIterable([1, 2, 3]));           // true
console.log(isIterable("Hello"));            // true
console.log(isIterable(new Map()));          // true
console.log(isIterable(new Set()));          // true
console.log(isIterable(new WeakMap()));      // false
console.log(isIterable(new WeakSet()));      // false
```

Функция `isIterable()` просто проверяет наличие итератора по умолчанию в объекте и является ли он функцией. Аналогичные проверки производит цикл `for-of` перед выполнением.

До сих пор в этом разделе демонстрировались примеры применения `Symbol.iterator` со встроенными итерируемыми типами, но свойство `Symbol.iterator` можно также использовать для создания собственных итерируемых объектов.

Создание итерируемых объектов

Объекты, определяемые разработчиком, по умолчанию не являются итерируемыми, но их можно сделать таковыми, добавив свойство `Symbol.iterator`, содержащее генератор. Например:

```
let collection = {
    items: [],
    *[Symbol.iterator]() {
        for (let item of this.items) {
            yield item;
        }
    }
};

collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}
```

Этот фрагмент выведет следующее:

```
1
2
3
```

В этом примере сначала определяется итератор по умолчанию для объекта `collection`. Итератор по умолчанию создается методом `Symbol.iterator`, который фактически является генератором (обратите внимание на звездочку перед именем метода). Генератор использует цикл `for-of` для итераций по значениям в массиве `this.items` и возвращает их по одному с помощью `yield`. Вместо управления итерациями вручную и извлечения значений для итератора по умолчанию объект `collection` полагается на итератор по умолчанию массива `this.items`.

ПРИМЕЧАНИЕ

В разделе «Делегирование генераторов» (см. ниже) описываются разные подходы к применению итераторов других объектов.

Теперь вы получили некоторое представление о том, как использовать итератор по умолчанию массивов, но в ECMAScript 6 существует много других встроенных итераторов, упрощающих работу с коллекциями данных.

Встроенные итераторы

Итераторы являются важным элементом ECMAScript 6, и поэтому нет необходимости создавать собственные итераторы для многих встроенных типов, поскольку они уже включены в язык. Вам придется определять собственные итераторы, только когда стандартные не будут соответствовать вашим целям — обычно при определении собственных объектов или классов. В остальных случаях вы сможете положиться на встроенные итераторы. Чаще других, пожалуй, вам придется использовать итераторы коллекций.

Итераторы коллекций

В ECMAScript 6 имеется три типа объектов коллекций: массивы, ассоциативные массивы и множества. Все три имеют следующие встроенные итераторы, помогающие в навигации по их содержимому:

- `entries()`. Возвращает итератор, значениями которого являются пары ключ/значение.
- `values()`. Возвращает итератор, значениями которого являются значения элементов коллекции.
- `keys()`. Возвращает итератор, значениями которого являются ключи, содержащиеся в коллекции.

Вызов этих методов позволяет получать итераторы для коллекций.

Итератор `entries()`

Итератор `entries()` возвращает двухэлементный массив в ответ на каждый вызов `next()`. Двухэлементный массив содержит ключ и значение для каждого элемента в коллекции. В первом элементе для массивов возвращается числовой индекс, для множеств — копия значения (потому что в множествах значения дублируют ключи), для ассоциативных массивов — ключ.

Ниже приводится несколько примеров использования итератора `entries()`:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

for (let entry of colors. entries()) {
  console.log(entry);
}
```

```
for (let entry of tracking.entries()) {  
    console.log(entry);  
}  
  
for (let entry of data.entries()) {  
    console.log(entry);  
}
```

Вызовы метода `console.log()` выведут следующее:

```
[0, "red"]  
[1, "green"]  
[2, "blue"]  
[1234, 1234]  
[5678, 5678]  
[9012, 9012]  
["title", "Understanding ECMAScript 6"]  
["format", "ebook"]
```

В этом фрагменте кода вызывается метод `entries()`, чтобы получить итератор для коллекции каждого вида, и используются циклы `for-of` для итераций по элементам. Вывод в консоль позволяет увидеть, как для каждого объекта возвращаются пары ключей и значений.

Итератор `values()`

Итератор `values()` просто возвращает значения в том виде, в каком они хранятся в коллекции. Например:

```
let colors = [ "red", "green", "blue" ];  
let tracking = new Set([1234, 5678, 9012]);  
let data = new Map();  
  
data.set("title", "Understanding ECMAScript 6");  
data.set("format", "ebook");  
  
for (let value of colors.values()) {  
    console.log(value);  
}  
  
for (let value of tracking.values()) {  
    console.log(value);  
}  
  
for (let value of data.values()) {  
    console.log(value);  
}
```

Этот фрагмент выведет следующее:

```
"red"  
"green"  
"blue"  
1234
```

```
5678
9012
"Understanding ECMAScript 6"
"ebook"
```

Вызов итератора `values()`, как в данном примере, возвращает данные из коллекции без дополнительной информации об их местоположении в коллекции.

Итератор `keys()`

Итератор `keys()` возвращает ключи, хранящиеся в коллекции. Для простых массивов он возвращает только числовые ключи и никогда не возвращает собственные свойства объекта массива. Для множеств ключи являются копиями значений, поэтому `keys()` и `values()` возвращают один и тот же итератор. Для ассоциативных массивов итератор `keys()` возвращает уникальные ключи. Следующий пример демонстрирует действие итератора для всех трех типов:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

for (let key of colors.keys()) {
    console.log(key);
}

for (let key of tracking.keys()) {
    console.log(key);
}

for (let key of data.keys()) {
    console.log(key);
}
```

Этот фрагмент выведет следующее:

```
0
1
2
1234
5678
9012
"title"
"format"
```

В этом примере в трех циклах `for-of` с помощью итератора `keys()` извлекаются ключи из `colors`, `tracking` и `data`, затем они выводятся. Для объекта обычного массива выводятся только числовые индексы, даже если вы добавите в него свои именованные свойства. Это отличает цикл `for-of` от `for-in` при работе с массивами — цикл `for-in` выполняет итерации по всем свойствам, а не только по числовым индексам.

Итераторы по умолчанию для типов коллекций

Каждый тип коллекций имеет свой итератор по умолчанию, который используется циклом `for-of`, если итератор не указан явно. Для массивов и множеств в качестве итератора по умолчанию используется метод `values()`, а для ассоциативных массивов — метод `entries()`. Эти умолчания немного упрощают использование коллекций в циклах `for-of`. Например, взгляните на следующий пример:

```
let colors = [ "red", "green", "blue" ];
let tracking = new Set([1234, 5678, 9012]);
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "print");

// действует так же, как при использовании colors.values()
for (let value of colors) {
    console.log(value);
}

// действует так же, как при использовании tracking.values()
for (let num of tracking) {
    console.log(num);
}

// действует так же, как при использовании data.entries()
for (let entry of data) {
    console.log(entry);
}
```

Когда итератор не указан явно, используется функция итератора по умолчанию. Итераторы по умолчанию для массивов, множеств и ассоциативных массивов устроены таким образом, что отражают, как были инициализированы эти объекты, поэтому предыдущий код выведет следующее:

```
"red"
"green"
"blue"
1234
5678
9012
["title", "Understanding ECMAScript 6"]
["format", "print"]
```

Массивы и множества по умолчанию возвращают свои значения, тогда как ассоциативные массивы возвращают результаты в виде тех же массивов, которые могли передаваться в конструктор `Map`. С другой стороны, множества и ассоциативные массивы со слабыми ссылками не имеют встроенных итераторов. Механизм управления слабыми ссылками не предполагает возможности знать точно, сколько значений хранится в таких коллекциях, что, в свою очередь, означает отсутствие возможности выполнять итерации по ним.

ДЕСТРУКТУРИЗАЦИЯ И ЦИКЛЫ FOR-OF

Поведение по умолчанию конструктора ассоциативных массивов также можно использовать в циклах `for-of` с деструктуризацией, как в следующем примере:

```
let data = new Map();

data.set("title", "Understanding ECMAScript 6");
data.set("format", "ebook");

// действует так же, как при использовании data.entries()
for (let [key, value] of data) {
    console.log(key + "=" + value);
}
```

Цикл `for-of` в этом примере использует деструктурированный массив для присваивания переменным `key` и `value` ключа и значения каждого элемента ассоциативного массива. Применяя такой подход, можно работать с ключами и значениями напрямую без необходимости извлекать их из двухэлементного массива или обращаться к ассоциативному массиву, чтобы получить ключ или значение. Использование деструктурированного массива для работы с ассоциативными массивами делает цикл `for-of` одинаково удобным для работы и с ассоциативными массивами, и с множествами, и с обычными массивами.

Итераторы строк

Строки в JavaScript еще больше стали похожи на массивы после выхода ECMAScript 5. Например, в ECMAScript 5 была формализована форма записи с квадратными скобками для обращения к символам в строках (то есть синтаксис `text[0]` для получения первого символа и т. д.). Но квадратные скобки работают с кодовыми единицами, а не с символами, поэтому с их помощью нельзя получить двухбайтный символ, как показывает следующий пример:

```
var message = "A 𐀀 B";

for (let i=0; i < message.length; i++) {
    console.log(message[i]);
}
```

В этом примере используются форма записи с квадратными скобками и свойство `length` для итераций и вывода символов Юникода, содержащихся в строке. Вывод получается немного неожиданным:

```
A
(пустая строка)
(пустая строка)
(пустая строка)
(пустая строка)
B
```

Поскольку двухбайтный символ в данном случае интерпретируется как две отдельные кодовые единицы, между `A` и `B` было выведено четыре пустые строки.

К счастью, в ECMAScript 6 была предпринята попытка обеспечить полноценную поддержку Юникода (как описывается в главе 2), и итератор по умолчанию для строк достаточно успешно решает проблему итераций по символам в строках. По сути, итератор по умолчанию работает с

символами, а не с кодовыми единицами. Если изменить предыдущий пример, задействовав в нем строковый итератор по умолчанию с циклом `for-of`, можно получить ожидаемый результат. Измененный пример приводится ниже:

```
var message = "A 𐄂 B";

for (let c of message) {
  console.log(c);
}
```

Этот фрагмент выведет следующее:

```
A
(пустая строка)
 𐄂(пустая строка)
B
```

Этот результат точнее соответствует ожиданиям при работе с символами: цикл благополучно вывел символ Юникода, как и все остальное.

Итераторы NodeList

В объектной модели документа (DOM) имеется тип `NodeList`, представляющий коллекцию элементов в документе. Для тех, кто пишет сценарии на JavaScript для выполнения в веб-браузерах, всегда было немного трудно осознать различия между объектами `NodeList` и массивами. Обе структуры, объекты `NodeList` и массивы, имеют свойство `length`, хранящее количество элементов, и при работе с обеими используются квадратные скобки для доступа к отдельным элементам. Однако внутренне `NodeList` и массив действуют совершенно по-разному, из-за чего разработчики часто допускают ошибки.

С появлением в ECMAScript 6 итераторов по умолчанию определение `NodeList` в DOM (дается в спецификации HTML, а не ECMAScript 6) включает итератор по умолчанию, который действует, подобно итератору по умолчанию массива. Это означает, что `NodeList` можно использовать в цикле `for-of` или в любом другом контексте, где применяется итератор по умолчанию. Например:

```
var divs = document.getElementsByTagName("div");

for (let div of divs) {
  console.log(div.id);
}
```

В этом коде вызывается `getElementsByTagName()`, чтобы получить `NodeList`, представляющий все элементы `<div>` в объекте `document`. Затем цикл `for-of` выполняет обход элементов и выводит значения их атрибутов `id`. В результате код получился очень похожим на код, который обрабатывал бы стандартный массив.

Оператор расширения и итерируемые объекты, не являющиеся массивами

В главе 7 рассказывалось, что оператор расширения (`...`) позволяет преобразовать множество в массив. Например:

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];
```

```
console.log(array);    // [1,2,3,4,5]
```

Этот фрагмент использует оператор расширения внутри литерала массива для заполнения этого массива значениями из множества `set`. Оператор расширения работает со всеми итерируемыми объектами и использует итератор по умолчанию, чтобы определить, какие значения включить в массив. Значения включаются в массив в том порядке, в каком они возвращаются итератором. Этот пример действует правильно, потому что множества являются итерируемыми объектами, но оператор расширения также хорошо работает с любыми итерируемыми объектами. Вот еще один пример:

```
let map = new Map([["name", "Nicholas"], ["age", 25]]),  
    array = [...map];  
  
console.log(array);    // [["name", "Nicholas"], ["age", 25]]
```

Здесь оператор расширения преобразует ассоциативный массив `map` в массив массивов. Так как итератор по умолчанию для ассоциативных массивов возвращает пары ключ/значение, получившийся в результате массив выглядит похожим на массив, который был передан в вызов `new Map()`.

Оператор расширения можно использовать в литерале массива столько раз, сколько потребуется, и везде, где необходимо вставить несколько элементов из итерируемого объекта. Эти элементы просто появятся в новом массиве там, где находится оператор расширения. Например:

```
let smallNumbers = [1, 2, 3],  
    bigNumbers = [100, 101, 102],  
    allNumbers = [0, ...smallNumbers, ...bigNumbers];  
  
console.log(allNumbers.length);    // 7  
console.log(allNumbers);           // [0, 1, 2, 3, 100, 101, 102]
```

Здесь оператор расширения использовался для создания массива `allNumbers` из значений в `smallNumbers` и `bigNumbers`. Значения помещаются в `allNumbers` в том же порядке, в каком массивы следуют в определении `allNumbers`: 0 — первое значение, за которым следуют значения из `smallNumbers`, а затем значения из `bigNumbers`. Исходные массивы при этом не изменяются, потому что их значения просто копируются в `allNumbers`.

Поскольку оператор расширения можно использовать с любыми итерируемыми объектами, он оказывается самым простым способом преобразования итерируемых объектов в массивы. С его помощью можно преобразовать строку в массив символов (не кодовых единиц) или объект `NodeList` в массив узлов.

Изучив основные возможности итераторов, включая `for-of` и оператор расширения, можно перейти к рассмотрению более сложных приемов применения итераторов.

Дополнительные возможности итераторов

Пользуясь основными возможностями итераторов и удобным способом их создания с применением генераторов, можно добиться многого. Однако итераторы позволяют также решать задачи, связанные не только с выполнением итераций по значениям коллекций. В ходе работы над спецификацией ECMAScript 6 появилось много уникальных идей и шаблонов программирования, подтолкнувших создателей к добавлению в язык еще большего количества возможностей. Некоторые из этих новшеств выглядят малозначащими, но все вместе они позволяют реализовать довольно интересные варианты взаимодействий, обсуждаемые в следующих разделах.

Передача аргументов в итераторы

На протяжении всей главы демонстрировались итераторы, возвращающие значения посредством метода `next()` или с использованием ключевого слова `yield` в генераторах. Но также есть возможность передавать аргументы в итераторы через метод `next()`. Когда метод `next()` получает аргумент, он становится значением, возвращаемым инструкцией `yield` внутри генератора. Эта особенность играет важную роль в реализации расширенных возможностей, таких как асинхронное программирование. Рассмотрим простой пример:

```
function *createIterator() {
  let first = yield 1;
  let second = yield first + 2;    // 4 + 2
  yield second + 3;                // 5 + 3
}

let iterator = createIterator();

console.log(iterator.next());      // "{ value: 1, done: false }"
console.log(iterator.next(4));    // "{ value: 6, done: false }"
console.log(iterator.next(5));    // "{ value: 8, done: false }"
console.log(iterator.next());     // "{ value: undefined, done: true }"
```

Первый вызов `next()` — особый случай, когда любой переданный аргумент будет потерян. Поскольку аргументы, переданные в `next()`, превращаются в значения, возвращаемые инструкцией `yield`, аргумент в первом вызове `next()` мог бы просто заменить первую инструкцию `yield` функции генератора, если бы он был доступен до этой инструкции `yield`. Однако это невозможно, поэтому бессмысленно передавать аргумент в первый вызов `next()`.

Во второй вызов `next()` передается аргумент со значением 4. В результате внутри функции генератора число 4 присваивается переменной `first`. В инструкции `yield`, включающей присваивание, правая часть выражения вычисляется в первом вызове `next()`, а левая часть — во втором вызове `next()`, прежде чем функция продолжит выполнение. Поскольку во второй вызов `next()` передается число 4, оно присваивается переменной `first`, и выполнение продолжается.

Вторая инструкция `yield` использует результат, возвращенный первой инструкцией `yield`, и прибавляет к нему 2, в результате чего в вызывающую программу возвращается число 6. Когда `next()` вызывается в третий раз, ему передается аргумент с числом 5. Это значение присваивается переменной `second` и затем используется в третьей инструкции `yield`, возвращающей 8.

Происходящее проще представить, если выделить код, который выполняется в каждом вызове функции генератора. На рис. 8.1 оттенками серого выделены фрагменты кода, выполняемого перед возвратом значений.

```
next()
next(4)
next(5)

function*createIterator(){
  let first = yield 1;
  let second = yield first + 2;
  yield second + 3;
}
```

Рис. 8.1: Порядок выполнения кода внутри генератора

Светло-серым фоном выделен первый вызов `next()` и соответствующий код, выполняющийся в генераторе. Серым фоном выделен вызов `next(4)` и код, выполняющийся в ответ на этот вызов. Темно-серым фоном выделен вызов `next(5)` и код, выполняющийся в результате этого вызова. Вся хитрость в том, что правая часть в каждом выражении выполняется и останавливается до того, как будет выполнена левая часть. Это делает функции генераторов более сложными в отладке по сравнению с простыми функциями.

Как видите, инструкция `yield` может действовать, подобно `return`, когда в метод `next()` передается значение, но это не единственный трюк, который можно проделывать внутри генератора. Имеется также возможность возбуждать ошибки внутри итераторов.

Возбуждение ошибок внутри итераторов

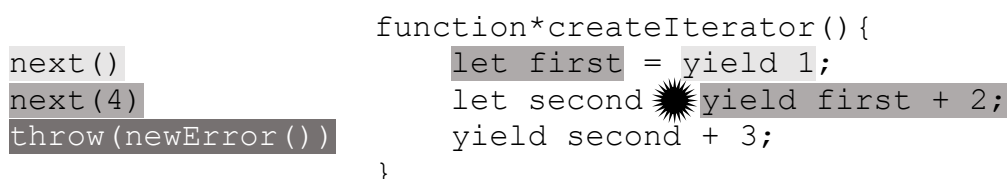
В итераторы можно передавать не только данные, но и ошибочные условия. Итераторы могут реализовать метод `throw()`, вызов которого требует от итератора возбудить ошибку, когда он возобновит выполнение. Эта особенность наиболее важна для асинхронного программирования, но также добавляет гибкости генераторам, когда требуется симитировать возврат значений и возбуждение ошибок (два способа выйти из функции). Вы можете передать в метод `throw()` объект ошибки, которая должна быть возбуждена, когда итератор продолжит работу. Например:

```
function *createIterator() {
  let first = yield 1;
  let second = yield first + 2;           // вернет 4 + 2 и возбудит ошибку
  yield second + 3;                     // никогда не будет выполнена
}

let iterator = createIterator();

console.log(iterator.next());             // "{ value: 1, done: false }"
console.log(iterator.next(4));           // "{ value: 6, done: false }"
console.log(iterator.throw(new Error("Boom"))); // ошибка, возбужденная генератором
```

В этом примере первые две инструкции `yield` выполняются как обычно, но после вызова `throw()` перед вычислением `let second` будет возбуждена ошибка. В результате работа кода прерывается, как если бы ошибка была возбуждена непосредственно. Единственное отличие — точка появления ошибки. На рис. 8.2 показано, какой код выполняется на каждом шаге.



```
function*createIterator(){
  next()      let first = yield 1;
  next(4)     let second = yield first + 2;
  throw(newError()) yield second + 3;
}
```

Рис. 8.2: Возбуждение ошибки внутри генератора

Как и на рис. 8.1, светло-серым и серым фоном выделены вызовы `next()` и `yield`, выполняемые без ошибок. Вызов `throw()` выделен темно-серым фоном, а черной звездой отмечена примерная точка возбуждения ошибки внутри генератора. Первые две инструкции `yield` выполнились, а после вызова `throw()` перед выполнением какого-либо другого кода была возбуждена ошибка. Зная это, можно перехватывать такие ошибки внутри генератора с помощью блока `try-catch`:

```
function *createIterator() {
  let first = yield 1;
  let second;

  try {
    second = yield first + 2; // вернет 4 + 2 и возбудит ошибку
  } catch (ex) {
    second = 6;               // в случае ошибки присваивает другое значение
  }
  yield second + 3;
}
```

```
let iterator = createIterator();

console.log(iterator.next());           // "{ value: 1, done: false }"
console.log(iterator.next(4));          // "{ value: 6, done: false }"
console.log(iterator.throw(new Error("Boom"))); // "{ value: 9, done: false }"
console.log(iterator.next());           // "{ value: undefined, done: true }"
```

В этом примере вторая инструкция `yield` заключена в блок `try-catch`. Эта инструкция выполняется без ошибок, но когда приходит черед присвоить значение переменной `second`, возбуждается ошибка. Блок `catch` перехватывает ее и присваивает переменной число 6. Далее поток выполнения достигает следующей инструкции `yield`, которая возвращает 9.

Обратите внимание на одну интересную особенность: метод `throw()` вернул объект результата, в точности как метод `next()`. Поскольку ошибка была перехвачена внутри генератора, код продолжил выполнение до следующей инструкции `yield` и вернул следующее значение — 9.

Это помогает интерпретировать методы `next()` и `throw()` как команды итератора. Метод `next()` командует итератору продолжить выполнение (возможно, с указанным значением), а метод `throw()` командует продолжить с возбуждением ошибки. Что произойдет после этого, зависит от реализации генератора.

Методы `next()` и `throw()` управляют выполнением внутри итератора, когда используется инструкция `yield`, но можно также воспользоваться инструкцией `return`. Однако `return` действует здесь немного иначе, чем в обычных функциях, как будет видно из следующего раздела.

Инструкции `return` в генераторах

Генераторы — это обычные функции, поэтому в них можно использовать инструкцию `return`, чтобы завершить работу генератора раньше и определить возвращаемое значение для последнего вызова метода `next()`. В большинстве примеров в этой главе последний вызов метода `next()` итератора возвращает `undefined`, но можно вернуть альтернативное значение с помощью `return`, как в обычных функциях.

Инструкция `return` в генераторе указывает, что все элементы обработаны, поэтому свойству `done` присваивается значение `true`, а свойству `value` — значение, указанное в инструкции `return`. Следующий пример просто завершает итерации раньше времени с помощью `return`:

```
function *createIterator() {
  yield 1;
  return;
  yield 2;
  yield 3;
}

let iterator = createIterator();

console.log(iterator.next()); // "{ value: 1, done: false }"
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

Здесь в генераторе за первой инструкцией `yield` следует инструкция `return`. Она сообщает, что все значения исчерпаны, поэтому остальные инструкции `yield` не выполняются (они оказываются недостижимыми).

Инструкции `return` можно передать значение, которое окажется в поле `value` возвращаемого объекта. Например:

```
function *createIterator() {
  yield 1;
  return 42;
}
```

```
}

let iterator = createIterator();

console.log(iterator.next()); // "{ value: 1, done: false }"
console.log(iterator.next()); // "{ value: 42, done: true }"
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

Здесь второй вызов `next()` (который оказывается первым, вернувшим `true` в поле `done`) возвращает число 42 в поле `value`. Третий вызов `next()` возвращает объект, свойство `value` которого вновь получает значение `undefined`. Значение, указанное в инструкции `return`, доступно в возвращаемом объекте только один раз, после чего все остальные вызовы `next()` будут возвращать объект со значением `undefined` в поле `value`.

ПРИМЕЧАНИЕ

Оператор расширения (...) и цикл `for-of` игнорируют значение, указанное в инструкции `return`, так как, обнаружив значение `true` в свойстве `done`, они прекращают читать свойство `value`. Однако значения, возвращаемые итератором с помощью `return`, могут пригодиться при делегировании генераторов.

Делегирование генераторов

Иногда может понадобиться объединить значения, возвращаемые двумя итераторами. Генераторы можно делегировать другим генераторам, используя специальную форму инструкции `yield` со звездочкой (*). Так же как в объявлениях генераторов, точное местоположение звездочки не имеет значения. Главное, чтобы она находилась между ключевым словом `yield` и именем функции генератора. Например:

```
function *createNumberIterator() {
  yield 1;
  yield 2;
}

function *createColorIterator() {
  yield "red";
  yield "green";
}

function *createCombinedIterator() {
  yield *createNumberIterator();
  yield *createColorIterator();
  yield true;
}

var iterator = createCombinedIterator();

console.log(iterator.next()); // "{ value: 1, done: false }"
console.log(iterator.next()); // "{ value: 2, done: false }"
console.log(iterator.next()); // "{ value: "red", done: false }"
console.log(iterator.next()); // "{ value: "green", done: false }"
console.log(iterator.next()); // "{ value: true, done: false }"
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

В этом примере генератор `createCombinedIterator()` делегирует работу сначала генератору `createNumberIterator()`, а затем генератору `createColorIterator()`. Снаружи возвращаемый итератор выглядит как один цельный итератор, который производит все значения. Каждый вызов `next()` делегируется соответствующему итератору, созданному с помощью `createNumberIterator()` или `createColorIterator()`, пока он не опустеет. Затем выполняется заключительная инструкция `yield`, возвращающая `true`.

Делегирование генераторов позволяет также использовать значения, возвращаемые генераторами с помощью инструкции `return`. Такой простой способ обращения к возвращаемым значениям может очень пригодиться для решения сложных задач. Например:

```
function *createNumberIterator() {
  yield 1;
  yield 2;
  return 3;
}

function *createRepeatingIterator(count) {
  for (let i=0; i < count; i++) {
    yield "repeat";
  }
}

function *createCombinedIterator() {
  let result = yield *createNumberIterator();
  yield *createRepeatingIterator(result);
}

var iterator = createCombinedIterator();

console.log(iterator.next()); // "{ value: 1, done: false }"
console.log(iterator.next()); // "{ value: 2, done: false }"
console.log(iterator.next()); // "{ value: "repeat", done: false }"
console.log(iterator.next()); // "{ value: "repeat", done: false }"
console.log(iterator.next()); // "{ value: "repeat", done: false }"
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

В этом примере генератор `createCombinedIterator()` делегирует работу генератору `createNumberIterator()` и присваивает возвращаемое им значение переменной `result`. Поскольку `createNumberIterator()` содержит инструкцию `return 3`, его возвращаемое значение равно 3. Затем переменная `result` передается в `createRepeatingIterator()` в виде аргумента, сообщаящего, сколько раз должна быть произведена одна и та же строка (в данном случае три раза).

Обратите внимание, что значение 3 не вернул ни один из вызовов метода `next()`. Оно существует только внутри генератора `createCombinedIterator()`. Однако его можно вывести, добавив еще одну инструкцию `yield`, например:

```
function *createNumberIterator() {
  yield 1;
  yield 2;
  return 3;
}

function *createRepeatingIterator(count) {
  for (let i=0; i < count; i++) {
```



```
        yield "repeat";
    }
}

function *createCombinedIterator() {
    let result = yield *createNumberIterator();
    yield result;
    yield *createRepeatingIterator(result);
}

var iterator = createCombinedIterator();

console.log(iterator.next()); // "{ value: 1, done: false }"
console.log(iterator.next()); // "{ value: 2, done: false }"
console.log(iterator.next()); // "{ value: 3, done: false }"
console.log(iterator.next()); // "{ value: "repeat", done: false }"
console.log(iterator.next()); // "{ value: "repeat", done: false }"
console.log(iterator.next()); // "{ value: "repeat", done: false }"
console.log(iterator.next()); // "{ value: undefined, done: true }"
```

В этом примере дополнительная инструкция `yield` явно выводит значение, возвращаемое генератором `createNumberIterator()`.

ПРИМЕЧАНИЕ

Конструкцию `yield *` можно применить непосредственно к строке (например: `yield * "hello"`). В результате будет использоваться итератор по умолчанию для строк.

Асинхронное выполнение заданий

Наиболее захватывающий аспект генераторов непосредственно связан с асинхронным программированием. Асинхронное программирование в JavaScript — это обоюдоострое оружие: реализация асинхронного выполнения простых задач не вызывает сложностей, но превращение сложной задачи в асинхронную может вылиться в целую эпопею. Поскольку генераторы позволяют приостанавливать выполнение кода, они открывают много возможностей, связанных с асинхронной обработкой.

Традиционный способ выполнения асинхронных операций заключается в вызове функции, которой передается функция обратного вызова. Например, рассмотрим процедуру чтения файла с диска в среде Node.js:

```
let fs = require("fs");

fs.readFile("config.json", function(err, contents) {
    if (err) {
        throw err;
    }

    doSomethingWith(contents);
    console.log("Done");
});
```

В вызов метода `fs.readFile()` передается имя файла и функция обратного вызова. Когда операция завершается, вызывается функция, переданная в аргументе. Она проверяет наличие ошибок

и в случае их отсутствия обрабатывает полученное содержимое в `contents`. Этот прием хорошо работает, когда в программе выполняется небольшое, конечное количество асинхронных заданий, но дело существенно усложняется, когда возникает необходимость использовать вложенные обратные вызовы или как-то иначе упорядочить последовательность асинхронных заданий. В таких ситуациях с успехом можно использовать генераторы и инструкцию `yield`.

Простой инструмент выполнения заданий

Так как `yield` останавливает выполнение и ждет следующего вызова метода `next()` перед продолжением, можно организовать асинхронное выполнение без использования обратных вызовов. Для начала нам понадобится функция, которая вызовет генератор и запустит итератор, например:

```
function run(taskDef) {

    // создать итератор
    let task = taskDef();

    // запустить задание
    let result = task.next();

    // рекурсивная функция, продолжающая вызывать next()
    function step() {

        // если работа продолжается
        if (!result.done) {
            result = task.next();
            step();
        }
    }

    // запустить обработку
    step();
}
```

Функция `run()` принимает определение задания (функцию-генератор). Она вызывает генератор, чтобы создать итератор, и сохраняет его в переменной `task`. Первый вызов `next()` запускает итератор и сохраняет результат для последующего использования. Функция `step()` сравнивает свойство `result.done` со значением `false` и, если условие выполняется, вызывает `next()` перед рекурсивным вызовом самой себя. Результат каждого вызова `next()` сохраняется в `result`, то есть содержимое этой переменной постоянно затирается последней информацией. Первый вызов `step()` запускает процесс проверки свойства `result.done`, чтобы выяснить, есть ли еще данные для обработки.

С помощью такой реализации `run()` можно запустить генератор, содержащий несколько инструкций `yield`, например:

```
run(function*() {
    console.log(1);
    yield;
    console.log(2);
    yield;
    console.log(3);
});
```

Этот пример просто выводит три числа в консоль с целью наглядно показать, что были выполнены все вызовы `next()`. Однако простой вывод нескольких фиксированных значений асинхронным способом не особенно полезен. Поэтому далее мы посмотрим, как передавать значения в итератор и получать их из итератора.

Выполнение заданий с данными

Самый простой способ передать данные через инструмент выполнения заданий — включить значение, указанное в инструкции `yield`, в следующий вызов метода `next()`. Для этого требуется только передать `result.value`, как показано ниже:

```
function run(taskDef) {

    // создать итератор
    let task = taskDef();

    // запустить задание
    let result = task.next();

    // рекурсивная функция, продолжающая вызывать next()
    function step() {

        // если работа продолжается
        if (!result.done) {
            result = task.next(result.value);
            step();
        }
    }

    // запустить обработку
    step();
}
```

Теперь, когда значение `result.value` передается в `next()` в виде аргумента, можно передавать данные между вызовами `yield`, например:

```
run(function*() {
    let value = yield 1;
    console.log(value);      // 1
    value = yield value + 3;
    console.log(value);      // 4
});
```

Этот пример выведет в консоль два числа: 1 и 4. Значение 1 получено в результате выполнения `yield 1`, потому что число 1 было тут же передано обратно в переменную `value`. Число 4 было вычислено сложением переменной `value` с числом 3 и передано обратно в переменную `value`. Теперь, когда данные свободно передаются между вызовами `yield`, нам осталось сделать всего один маленький шаг, чтобы обеспечить возможность асинхронных вызовов.

Инструмент асинхронного выполнения заданий

Предыдущий пример демонстрировал передачу статических данных между вызовами `yield`, но ожидание в асинхронной обработке — это нечто иное. Инструмент выполнения заданий должен

знать об обратных вызовах и уметь их использовать. Поскольку выражения `yield` передают свои значения в инструмент выполнения заданий, любые вызовы функций должны возвращать значение, указывающее тем или иным образом, что вызов является асинхронной операцией, завершения которой инструмент выполнения заданий должен дожидаться.

Ниже демонстрируется один из способов сообщить, что значение представляет асинхронную операцию:

```
function fetchData() {
  return function(callback) {
    callback(null, "Hi!");
  };
}
```

Для целей этого примера нам потребуется функция, вызываемая инструментом выполнения заданий, которая будет возвращать функцию, выполняющую обратный вызов. Функция `fetchData()` возвращает функцию, которая принимает функцию обратного вызова в аргументе. Когда будет вызвана возвращаемая функция, она выполнит функцию обратного вызова с единственным фрагментом данных (строкой "Hi!"). Аргумент `callback` должен передаваться из инструмента выполнения заданий, чтобы гарантировать корректное взаимодействие функции обратного вызова с итератором. Несмотря на то что `fetchData()` является синхронной функцией, ее легко можно превратить в асинхронную, запустив функцию обратного вызова с небольшой задержкой, как показано ниже:

```
function fetchData() {
  return function(callback) {
    setTimeout(function() {
      callback(null, "Hi!");
    }, 50);
  };
}
```

Эта версия `fetchData()` вводит задержку в 50 мс перед запуском функции обратного вызова и демонстрирует, что данный шаблон хорошо подходит для выполнения не только синхронных, но и асинхронных операций. От вас требуется только гарантировать, что каждая функция, которая должна вызываться с помощью `yield`, следует одному и тому же шаблону.

Хорошо понимая, как функция может сообщить, что она выполняет обработку асинхронно, можно изменить инструмент выполнения заданий, чтобы учесть это обстоятельство. Всякий раз когда `result.value` представляет функцию, инструмент выполнения заданий будет выполнять ее вместо передачи в вызов метода `next()`. Ниже приводится обновленный код:

```
function run(taskDef) {

  // создать итератор
  let task = taskDef();

  // запустить задание
  let result = task.next();

  // рекурсивная функция, продолжающая вызывать next()
  function step() {

    // если работа продолжается
    if (!result.done) {
      if (typeof result.value === "function") {
```

```
        result.value(function(err, data) {
            if (err) {
                result = task.throw(err);
                return;
            }

            result = task.next(data);
            step();
        });
    } else {
        result = task.next(result.value);
        step();
    }
}

// запустить обработку
step();
}
```

Когда в свойстве `result.value` оказывается функция (проверяется оператором `===`), она запускается с функцией обратного вызова. Эта функция обратного вызова следует соглашению, принятому в Node.js, в соответствии с которым в первом аргументе ей передается любая возможная ошибка (`err`), а во втором аргументе — результат. Если аргумент `err` содержит непустое значение, это означает, что возникла ошибка, и вместо `task.next()` вызывается `task.throw()` с объектом ошибки, благодаря чему ошибка возбуждается в правильном месте. Если ошибка отсутствует, данные из аргумента `data` передаются в `task.next()` и результат сохраняется. Затем вызывается `step()`, чтобы продолжить обработку. Если свойство `result.value` содержит значение, не являющееся функцией, метод `next()` вызывается непосредственно.

Эта новая версия инструмента выполнения заданий готова обслуживать любые асинхронные задания. Чтобы прочитать данные из файла в среде Node.js, нужно создать обертку вокруг `fs.readFile()` и вернуть функцию, похожую на функцию `fetchData()`, представленную в начале раздела. Например:

```
let fs = require("fs");

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}
```

Метод `readFile()` принимает единственный аргумент, имя файла, и возвращает функцию, которая запускает функцию обратного вызова. Функция обратного вызова передается непосредственно в вызов метода `fs.readFile()`, который выполнит функцию обратного вызова по завершении. Теперь это задание можно запустить с помощью `yield`, как показано ниже:

```
run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

Этот фрагмент выполнит асинхронную операцию `readFile()` без каких-либо обратных вызовов, видимых в основном коде. Помимо инструкции `yield` код выглядит как синхронный. Если все функции, выполняющие асинхронные операции, будут следовать общему интерфейсу, вы сможете писать логику, которая выглядит как синхронный код.

Конечно, шаблон, представленный в примерах выше, имеет свои недостатки, а именно не всегда можно быть уверенным, что функция, возвращающая функцию, является асинхронной. Но сейчас вам важно лишь понять теорию, на которой основано выполнение заданий. Другой новой особенностью в ECMAScript 6 являются объекты `promise`, предлагающие более гибкие способы планирования асинхронных заданий, но подробнее об этом мы поговорим в главе 11.

В заключение

Итераторы — важный элемент ECMAScript 6 и фундамент некоторых других ключевых особенностей языка. С внешней стороны итераторы позволяют организовать получение последовательности значений с использованием простого API. Однако в ECMAScript 6 поддерживаются более сложные способы применения итераторов.

Для определения итераторов объектов применяется символ `Symbol.iterator`. Любые объекты, встроенные и созданные разработчиком, могут использовать этот символ для реализации метода, возвращающего итератор. Если объект поддерживает свойство `Symbol.iterator`, он считается итерируемым.

Для обхода итерируемых объектов и получения последовательности значений применяется цикл `for-of`. Цикл `for-of` проще в использовании, чем традиционный цикл `for`, потому что не требует следить за значениями и управлять моментом завершения цикла. Цикл `for-of` автоматически читает все значения из итератора, пока не прочитает их все, и затем завершается.

Чтобы упростить применение цикла `for-of`, многие объекты в ECMAScript 6 обладают итераторами по умолчанию. Все типы коллекций, — то есть массивы, простые и ассоциативные, и множества, — обладают итераторами, упрощающими доступ к их содержимому. Строки также имеют итератор по умолчанию, который упрощает выполнение итераций по символам в строке (именно по символам, а не по кодовым единицам).

Оператор расширения может применяться к любым итерируемым объектам и легко преобразовывать их в массивы. При преобразовании значения читаются из итератора и по отдельности вставляются в массив.

Генератор — это особая функция, которая автоматически создает и возвращает итератор. Определение генератора отличается наличием звездочки (*) и использованием ключевого слова `yield`, которое определяет, какое значение должно возвращаться в ответ на каждый вызов метода `next()`.

Возможность делегирования генераторов способствует более полной инкапсуляции поведения итератора, позволяя повторно использовать имеющиеся генераторы в новых генераторах. Существующий генератор можно задействовать внутри другого генератора, вызвав `yield*` вместо `yield`. Это дает возможность создавать итераторы, возвращающие значения из нескольких других итераторов.

Пожалуй, самым интересным и захватывающим аспектом генераторов и итераторов является возможность реализации асинхронных операций с использованием кода, имеющего линейный, синхронный вид. Вместо повсеместного использования функций обратного вызова можно написать код, который выглядит как синхронный, но использует инструкцию `yield` для ожидания завершения асинхронной операции.

Введение в классы JavaScript

В отличие от большинства языков программирования, официально считающихся объектно-ориентированными языками, JavaScript первоначально не поддерживал классы и классическое наследование как основной способ определения подобных и родственных объектов. Это вносило путаницу в умы многих разработчиков, и в период действия спецификаций, начиная с предшествовавших ECMAScript 1 и по ECMAScript 5, появилось большое количество библиотек, помогающих писать код на JavaScript, обладающий типичными объектно-ориентированными чертами. Несмотря на то что некоторые разработчики на JavaScript не чувствовали острой необходимости в классах, огромное количество библиотек, созданных специально с этой целью, подтолкнуло к включению классов в ECMAScript 6.

Приступая к исследованию классов ECMAScript 6, важно понимать, на каких механизмах они основаны, поэтому данная глава начинается с обсуждения приемов, с помощью которых разработчики на ECMAScript 5 добивались сходства программного кода с традиционными классами. Однако, как вы увидите далее, классы в ECMAScript 6 не точно соответствуют классам в других языках. Они имеют свои уникальные черты, обусловленные динамической природой JavaScript.

Структуры в ECMAScript 5, подобные классам

Как отмечалось выше, в ECMAScript 5 и более ранних версиях JavaScript не имел классов. Ближайшим эквивалентом определения классов было создание конструкторов с последующим добавлением методов в их прототипы и их использование для создания пользовательских типов. Например:

```
function PersonType(name) {
    this.name = name;
}

PersonType.prototype.sayName = function() {
    console.log(this.name);
}

var person = new PersonType("Nicholas");
person.sayName(); // выведет "Nicholas"

console.log(person instanceof PersonType); // true
console.log(person instanceof Object);    // true
```

В этом примере `PersonType` — это функция-конструктор, которая создает единственное свойство с именем `name`. Метод `sayName()` добавляется в прототип, поэтому одна и та же функция может совместно использоваться всеми экземплярами объекта `PersonType`. Затем с помощью оператора `new` создается новый экземпляр `PersonType`. Получившийся в результате объект `person` интерпретируется как экземпляр `PersonType` и `Object` через наследование прототипов.

Этот простой шаблон лежит в основе множества имитаций классов, реализованных в библиотеках JavaScript, и с этого момента мы начинаем исследование классов ECMAScript 6.

Объявление класса

Простейшей формой класса в ECMAScript 6 является объявление класса, которое выглядит похожим на объявления классов в других языках.

Объявление простого класса

Объявление класса начинается с ключевого слова `class`, за которым следует имя класса. Остальная часть объявления напоминает краткий способ объявления методов в литералах объектов, с той лишь разницей, что не требует ставить запятые между элементами класса. Ниже приводится пример объявления простого класса:

```
class PersonClass {

    // эквивалент конструктора PersonType
    constructor(name) {
        this.name = name;
    }

    // эквивалент метода PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
}

let person = new PersonClass("Nicholas");
person.sayName(); // выведет "Nicholas"

console.log(person instanceof PersonClass); // true
console.log(person instanceof Object); // true

console.log(typeof PersonClass); // "function"
console.log(typeof PersonClass.prototype.sayName); // "function"
```

Класс `PersonClass` действует подобно объекту `PersonType` из предыдущего примера. Но вместо определения функции как конструктора объявление класса позволяет определить конструктор непосредственно внутри класса как метод со специальным именем `constructor`. Поскольку для определения методов класса применяется сокращенный синтаксис, отпадает необходимость использовать ключевое слово `function`. Имена всех остальных методов не имеют специального значения, и допускается добавлять столько методов, сколько потребуется.

Собственные свойства, то есть свойства, принадлежащие экземпляру, а не прототипу, могут создаваться только внутри конструктора или методов класса. В этом примере `name` является собственным свойством. Я рекомендую создавать все собственные свойства внутри функции-конструктора, чтобы все определения были сосредоточены в одном месте внутри класса.

Самое интересное, что объявление класса — это лишь синтаксический сахар, скрывающий сложности описания пользовательского типа. Объявление `PersonClass` фактически создает функцию, действующую как метод `constructor`, что объясняет, почему `typeof PersonClass` возвращает результат `"function"`. Метод `sayName()` также в конечном итоге оказывается методом в `PersonClass.prototype`, подобно методу `sayName()` прототипа `PersonType.prototype` в предыдущем примере. Такое сходство позволяет смешивать пользовательские типы и классы, не заботясь об особенностях их использования.

ПРИМЕЧАНИЕ

Прототипы классов, такие как `PersonClass.prototype` в предыдущем примере, доступны только для чтения. Это означает, что свойству `prototype` нельзя присвоить новое значение, как это допускается в случае с функциями.

В чем преимущества синтаксиса определения классов?

Несмотря на сходство классов и пользовательских типов, они имеют важные отличия, о которых следует помнить:

- Объявления классов, в отличие от объявлений функций, не поднимаются в начало вещающего блока. Объявления классов действуют подобно объявлениям `let`, поэтому они хранятся во временной мертвой зоне, пока поток выполнения не достигнет их.
- Весь код в объявлении класса автоматически выполняется в строгом режиме. Внутри классов нет никакой возможности выйти из строгого режима.
- Все методы являются неперечислимыми. Это важное отличие от пользовательских типов, где требуется использовать `Object.defineProperty()`, чтобы сделать метод неперечислимым.
- У всех методов отсутствует внутренний метод `[[Construct]]`, и при попытке вызвать их с ключевым словом `new` возбуждается ошибка.
- Вызов конструктора класса без ключевого слова `new` завершается ошибкой.
- Попытка затереть имя класса внутри метода завершается ошибкой.

Учитывая все эти отличия, объявление `PersonClass` в предыдущем примере прямо эквивалентно следующему коду, не использующему синтаксис классов:

```
// прямой эквивалент PersonClass
let PersonType2 = (function() {

  "use strict";

  const PersonType2 = function(name) {

    // гарантировать возможность вызова функции
    // только с ключевым словом new
    if (typeof new.target === "undefined") {
      throw new Error("Constructor must be called with new.");
    }

    this.name = name;
  }
}
```

```
Object.defineProperty(PersonType2.prototype, "sayName", {
  value: function() {

    // гарантировать невозможность вызова метода
    // с ключевым словом new
    if (typeof new.target !== "undefined") {
      throw new Error("Method cannot be called with new.");
    }

    console.log(this.name);
  },
  enumerable: false,
  writable: true,
  configurable: true
});

return PersonType2;
})();
```

Первое, на что следует обратить внимание, — это два объявления `PersonType2`: `let`-объявление снаружи и `const`-объявление внутри выражения немедленно вызываемой функции (Immediately Invoked Function Expression, IIFE) — именно так предотвращается возможность затирания имени класса внутри его методов, тогда как код за пределами класса может сделать это. Функция-конструктор проверяет свойство `new.target`, чтобы убедиться, что она вызвана с ключевым словом `new`; в противном случае возбуждается ошибка. Далее метод `sayName()` определяется как неперечислимый и проверяет свойство `new.target`, чтобы убедиться, что он вызван без ключевого слова `new`. В заключение функция возвращает функцию-конструктор.

Этот пример показывает, что классы можно создавать и без применения нового синтаксиса, но он существенно упрощает эту задачу.

НЕИЗМЕНЯЕМЫЕ ИМЕНА КЛАССОВ

Имя класса является константой только внутри класса. Это означает, что его можно затереть извне класса, но не внутри метода класса. Например:

```
class Foo {
  constructor() {
    Foo = "bar";    // возбудит ошибку во время выполнения...
  }
}

// но после объявления класса это допустимо
Foo = "baz";
```

В данном примере `Foo` внутри конструктора класса — это отдельная привязка, отличная от привязки `Foo` за пределами класса. Внутренняя привязка `Foo` определяется как константа и не может быть затерта. Если конструктор попытается затереть привязку `Foo` любым значением, интерпретатор возбудит ошибку. Но так как внешняя привязка `Foo` определена с помощью `let`-объявления, внешний код может затереть ее в любой момент.

Классы-выражения

Классы, как и функции, имеют две формы: объявления и выражения. Объявления функций и классов начинаются с ключевого слова (`function` и `class` соответственно), за которым следует идентификатор. Функции имеют форму выражения, не требующую указывать идентификатор после `function`, а классы имеют форму выражения, не требующую указывать идентификатор после `class`. Такие *классы-выражения* предназначены для использования в объявлениях переменных или для передачи в функции через аргументы.

Простой класс-выражение

Ниже приводится класс-выражение, эквивалентный предыдущим примерам `PersonClass`, за которым следует некоторый код, использующий его:

```
let PersonClass = class {

    // эквивалент конструктора PersonType
    constructor(name) {
        this.name = name;
    }

    // эквивалент PersonType.prototype.sayName
    sayName() {
        console.log(this.name);
    }
};

let person = new PersonClass("Nicholas");
person.sayName(); // выведет "Nicholas"

console.log(person instanceof PersonClass); // true
console.log(person instanceof Object); // true

console.log(typeof PersonClass); // "function"
console.log(typeof PersonClass.prototype.sayName); // "function"
```

Как показывает этот пример, классы-выражения действительно не требуют наличия идентификатора после ключевого слова `class`. Кроме синтаксиса, во всем остальном классы-выражения функционально эквивалентны объявлениям классов. В анонимных классах-выражениях, как в предыдущем примере, свойству `PersonClass.name` присваивается пустая строка. В объявлении класса свойство `PersonClass.name` получило бы строковое значение `"PersonClass"`.

Выбор между объявлениями классов и классами-выражениями фактически является вопросом стиля программирования. В отличие от объявлений функций и функций-выражений, объявления классов и классы-выражения не поднимаются в начало вещающего блока, поэтому ваш выбор почти не повлияет на поведение кода во время выполнения. Единственное существенное отличие заключается в том, что анонимные классы-выражения содержат пустую строку в свойстве `name`, тогда как объявленные классы всегда хранят в свойстве `name` строку с именем класса (например, свойство `PersonClass.name` объявленного класса `PersonClass` получит значение `"PersonClass"`).

Именованные классы-выражения

В предыдущем примере был показан анонимный класс-выражение, но классам-выражениям, так же как функциям-выражениям, можно присваивать имена. Для этого нужно лишь включить идентификатор после ключевого слова `class`, например:

```
let PersonClass = class PersonClass2 {  
  
    // эквивалент конструктора PersonType  
    constructor(name) {  
        this.name = name;  
    }  
  
    // эквивалент PersonType.prototype.sayName  
    sayName() {  
        console.log(this.name);  
    }  
};  
  
console.log(typeof PersonClass);    // "function"  
console.log(typeof PersonClass2);  // "undefined"
```

В этом примере классу-выражению присвоено имя `PersonClass2`. Идентификатор `PersonClass2` существует только внутри определения класса, поэтому может использоваться в методах класса (таких, как `sayName()`). Выражение `typeof PersonClass2` за пределами класса вернуло значение `"undefined"`, потому что там привязка `PersonClass2` отсутствует. Чтобы понять причину этого, рассмотрим эквивалентное объявление без применения нового синтаксиса классов:

```
// прямой эквивалент именованного класса-выражения PersonClass  
let PersonClass = (function() {  
  
    "use strict";  
  
    const PersonClass2 = function(name) {  
  
        // гарантировать возможность вызова функции  
        // только с ключевым словом new  
        if (typeof new.target === "undefined") {  
            throw new Error("Constructor must be called with new.");  
        }  
  
        this.name = name;  
    }  
  
    Object.defineProperty(PersonClass2.prototype, "sayName", {  
        value: function() {  
  
            // гарантировать невозможность вызова метода  
            // с ключевым словом new  
            if (typeof new.target !== "undefined") {  
                throw new Error("Method cannot be called with new.");  
            }  
  
            console.log(this.name);  
        },  
        enumerable: false,  
        writable: true,  
        configurable: true  
    });  
});
```

```
    return PersonClass2;
  }());
```

Создание именованного класса-выражения немного меняет происходящее в недрах движка JavaScript. Для объявлений классов внешняя привязка (определяемая с помощью `let`) получает то же имя, что и внутренняя (определяемая с помощью `const`). В именованных классах-выражениях имя класса используется только для внутренней привязки в определении `const`, поэтому имя `PersonClass2` определено только внутри класса.

Несмотря на то что именованные классы-выражения действуют немного иначе, чем именованные функции-выражения, между ними все еще достаточно много сходства. И те и другие можно использовать в роли значений, и это открывает новые возможности, о которых рассказывается далее.

Классы как сущности первого класса

В программировании *сущностями первого класса* (*first-class citizen*) называют значения, которые можно передавать в функции, возвращать из функций и присваивать переменным. Функции в языке JavaScript — это сущности первого класса (их также называют *функциями первого класса*), и они являются одной из особенностей, придающих уникальность языку JavaScript.

Спецификация ECMAScript 6 продолжила эту традицию, сделав классы также сущностями первого класса и позволив использовать их множеством разных способов. Например, их можно передавать функциям в аргументах:

```
function createObject(classDef) {
    return new classDef();
}
```

```
let obj = createObject(class {

    sayHi() {
        console.log("Hi!");
    }

});
```

```
obj.sayHi();    // "Hi!"
```

Этот код вызывает функцию `createObject()` и передает ей анонимный класс-выражение. Функция создает экземпляр класса с помощью оператора `new` и возвращает его. Полученный экземпляр затем присваивается переменной `obj`.

Другой пример использования классов-выражений — создание синглтонов путем немедленно-го вызова конструктора класса. Для этого нужно добавить ключевое слово `new` перед классом-выражением и круглые скобки после. Например:

```
let person = new class {

    constructor(name) {
        this.name = name;
    }

    sayName() {
        console.log(this.name);
    }

}("Nicholas");

person.sayName();    // "Nicholas"
```

Здесь создается анонимный класс-выражение и сразу же вызывается его конструктор. Этот шаблон позволяет использовать синтаксис классов для создания синглтонов, не оставляя ссылку на класс доступной для остального кода. Круглые скобки в конце класса-выражения указывают, что здесь вызывается функция, и дают возможность передать аргументы.

Примеры, что приводились выше, демонстрировали классы с методами. Но классы позволяют также добавлять свойства с методами доступа с использованием синтаксиса, похожего на синтаксис литералов объектов.

Свойства с методами доступа

Все собственные свойства класса желательно определять внутри конструктора, тем не менее классы позволяют определять свойства с методами доступа в прототипе. Методы чтения (getter) определяются с помощью ключевого слова `get`, за которым следует пробел и идентификатор. Методы записи (setter) определяются с использованием ключевого слова `set`, как показано ниже:

```
class CustomHTMLElement {  
  
  constructor(element) {  
    this.element = element;  
  }  
  
  get html() {  
    return this.element.innerHTML;  
  }  
  
  set html(value) {  
    this.element.innerHTML = value;  
  }  
}  
  
var descriptor = Object.getOwnPropertyDescriptor(CustomHTMLElement.prototype,  
"html");  
  
console.log("get" in descriptor);    // true  
console.log("set" in descriptor);    // true  
console.log(descriptor.enumerable);  // false
```

В этом примере определяется класс `CustomHTMLElement`, служащий оберткой для существующего элемента DOM. Он объявляет свойство `html` с методами чтения/записи, которые фактически делегируют свою работу свойству `innerHTML` элемента. Свойство с методами доступа создается в `CustomHTMLElement.prototype` и, как любые другие методы, объявляется неперечислимым. Ниже показано, как выглядит эквивалентная реализация без использования синтаксиса классов:

```
// прямой эквивалент предыдущего примера  
let CustomHTMLElement = (function() {  
  
  "use strict";  
  
  const CustomHTMLElement = function(element) {  
  
    // гарантировать возможность вызова функции  
    // только с ключевым словом new
```

```
    if (typeof new.target === "undefined") {
        throw new Error("Constructor must be called with new.");
    }

    this.element = element;
}

Object.defineProperty(CustomHTMLElement.prototype, "html", {
    enumerable: false,
    configurable: true,
    get: function() {
        return this.element.innerHTML;
    },
    set: function(value) {
        this.element.innerHTML = value;
    }
});

return CustomHTMLElement;
})();
```

Как и предыдущие, этот пример демонстрирует, как много кода позволяет опустить синтаксис классов. Определение свойства `html` с методами доступа — единственный фрагмент, размер которого практически не изменился по сравнению с эквивалентным объявлением класса.

Вычисляемые имена членов

Литералы объектов и классы имеют еще несколько сходных черт. Методы классов и свойства с методами доступа могут иметь вычисляемые имена. Вместо идентификатора можно использовать выражение в квадратных скобках, как в синтаксисе литералов объектов. Например:

```
let methodName = "sayName";

class PersonClass {

    constructor(name) {
        this.name = name;
    }

    [methodName]() {
        console.log(this.name);
    }
};

let me = new PersonClass("Nicholas");
me.sayName();    // "Nicholas"
```

Эта версия объявления `PersonClass` использует переменную для определения имени метода внутри объявления. Строка `"sayName"` присваивается переменной `methodName`, а затем `methodName` используется в объявлении метода. Далее выполняется непосредственное обращение к методу `sayName()`.

Для объявления свойств с методами доступа также можно использовать вычисляемые имена, например:

```
let propertyName = "html";

class CustomHTMLElement {

  constructor(element) {
    this.element = element;
  }

  get [propertyName]() {
    return this.element.innerHTML;
  }

  set [propertyName](value) {
    this.element.innerHTML = value;
  }
}
```

Здесь методы чтения и записи для свойства `html` определяются с помощью переменной `propertyName`. Обращение к свойству по имени `.html` лишь отражает определение.

Вы познакомились с некоторыми сходными чертами классов и литералов объектов, включая методы, свойства с методами доступа и вычисляемые имена. Но есть еще одна общая черта, о которой следует упомянуть, — это генераторы.

Методы-генераторы

Как рассказывалось в главе 8, чтобы определить генератор в литерале объекта, нужно добавить звездочку (*) перед именем метода. Тот же синтаксис можно использовать в объявлениях классов, чтобы создать метод-генератор. Например:

```
class MyClass {

  *createIterator() {
    yield 1;
    yield 2;
    yield 3;
  }
}

let instance = new MyClass();
let iterator = instance.createIterator();
```

В этом фрагменте создается класс с именем `MyClass`, включающий метод-генератор `createIterator()`. Он возвращает итератор, значения которого жестко определены в генераторе. Методы-генераторы могут пригодиться в объектах, представляющих коллекции значений, когда желательно иметь простую возможность выполнять итерации по ним. Массивы, множества и ассоциативные массивы — все эти объекты имеют по несколько методов-генераторов, реализующих разные подходы к взаимодействиям с их элементами.

Методы-генераторы — удобный инструмент, но еще удобнее, если класс, представляющий коллекцию значений, определяет итератор по умолчанию. Чтобы определить итератор по умолчанию для класса, нужно использовать `Symbol.iterator` в определении метода-генератора:


```
class Collection {

    constructor() {
        this.items = [];
    }

    *[Symbol.iterator]() {
        yield *this.items.values();
    }
}

var collection = new Collection();
collection.items.push(1);
collection.items.push(2);
collection.items.push(3);

for (let x of collection) {
    console.log(x);
}

// Вывод:
// 1
// 2
// 3
```

В этом примере определяется метод-генератор с помощью синтаксиса вычисляемых имен, который делегирует основную работу итератору `values()` массива `this.items`. Любой класс, управляющий коллекцией значений, должен включать итератор по умолчанию, потому что некоторые операции с коллекциями требуют наличия итератора в коллекциях, которыми они оперируют. Теперь вы сможете использовать любой экземпляр `Collection` непосредственно в цикле `for-of` или в операторе расширения (`...`).

Добавление методов и свойств с методами доступа в прототип класса удобно, когда они должны быть доступны в экземплярах объекта. Но если вам необходимы методы или свойства с методами доступа, принадлежащие самому классу, вы должны использовать статические члены.

Статические члены

Еще один распространенный шаблон в ECMAScript 5 и предыдущих версиях — добавление методов непосредственно в конструкторы для имитации статических членов. Например:

```
function PersonType(name) {
    this.name = name;
}

// статический метод
PersonType.create = function(name) {
    return new PersonType(name);
}

// метод экземпляра
PersonType.prototype.sayName = function() {
    console.log(this.name);
}
```

```
};  
  
var person = PersonType.create("Nicholas");
```

В других языках программирования фабричный метод `PersonType.create()` мог бы считаться статическим, потому что он не зависит ни от какого экземпляра `PersonType`.

В ECMAScript 6 появился упрощенный способ создания статических членов — с использованием аннотации `static` перед методом или именем свойства с методами доступа. Например, ниже определяется класс, эквивалентный предыдущему примеру:

```
class PersonClass {  
  
    // эквивалент конструктора PersonType  
    constructor(name) {  
        this.name = name;  
    }  
  
    // эквивалент PersonType.prototype.sayName  
    sayName() {  
        console.log(this.name);  
    }  
  
    // эквивалент PersonType.create  
    static create(name) {  
        return new PersonClass(name);  
    }  
}  
  
let person = PersonClass.create("Nicholas");
```

Определение класса `PersonClass` включает единственный статический метод `create()`. Для его определения используется тот же синтаксис, что и для определения метода `sayName()`, за исключением ключевого слова `static`. Ключевое слово `static` можно применять к любым определениям методов и свойств с методами доступа внутри класса. Единственное ограничение — `static` нельзя использовать в определении метода `constructor`.

ПРИМЕЧАНИЕ

Статические члены недоступны как методы и свойства экземпляров. Для обращения к ним всегда необходимо использовать сам класс.

Наследование в производных классах

До выхода ECMAScript 6 реализация наследования в пользовательских типах была весьма трудоемким процессом. Для надлежащего оформления наследования требовалось выполнить несколько шагов. Например, взгляните на следующий фрагмент:

```
function Rectangle(length, width) {  
    this.length = length;  
    this.width = width;  
}  
  
Rectangle.prototype.getArea = function() {
```

```
        return this.length * this.width;
    };

function Square(length) {
    Rectangle.call(this, length, length);
}

Square.prototype = Object.create(Rectangle.prototype, {
    constructor: {
        value: Square,
        enumerable: true,
        writable: true,
        configurable: true
    }
});

var square = new Square(3);

console.log(square.getArea());           // 9
console.log(square instanceof Square);    // true
console.log(square instanceof Rectangle); // true
```

Объект `Square` наследует `Rectangle`, и, чтобы оформить это наследование, нужно записать в свойство `Square.prototype` ссылку на новый объект, созданный из `Rectangle.prototype`, а также вызвать метод `Rectangle.call()`. Такая последовательность действий часто приводила в замешательство начинающих осваивать JavaScript и была источником ошибок для опытных разработчиков.

Классы упростили реализацию наследования за счет знакомого ключевого слова **extends**, определяющего функцию, которую должен наследовать класс. Корректировка прототипа теперь выполняется автоматически, и вы можете обращаться к конструктору базового класса, вызывая метод `super()`. Ниже приводится эквивалент предыдущего примера на ECMAScript 6:

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }
}

class Square extends Rectangle {
    constructor(length) {

        // эквивалент Rectangle.call(this, length, length)
        super(length, length);
    }
}

var square = new Square(3);
```

```
console.log(square.getArea());           // 9
console.log(square instanceof Square);    // true
console.log(square instanceof Rectangle); // true
```

На этот раз наследование класса `Rectangle` оформлено в классе `Square` с помощью ключевого слова `extends`. Конструктор `Square` использует `super()` для вызова конструктора `Rectangle` с заданными аргументами. Обратите внимание, что, в отличие от версии для ECMAScript 5, идентификатор `Rectangle` в этом примере используется только внутри объявления класса (после `extends`).

Классы, наследующие другие классы, называют *производными классами*. Производные классы должны использовать `super()` в конструкторе; если этого не сделать, возникнет ошибка. Если вы решили не определять конструктор в производном классе, функция `super()` будет вызвана автоматически со всеми аргументами, указанными в инструкции создания нового экземпляра класса. Например, следующие два класса являются идентичными:

```
class Square extends Rectangle {
    // конструктор не определен
}

// эквивалентный класс
class Square extends Rectangle {
    constructor(...args) {
        super(...args);
    }
}
```

Второй класс в этом примере демонстрирует эквивалентную реализацию конструктора по умолчанию для всех производных классов. Все полученные аргументы передаются в том же порядке конструктору базового класса. В данном случае реализация получилась не совсем корректной, потому что конструктору `Square` требуется только один аргумент, поэтому в таких ситуациях лучше определять конструктор вручную.

ПРИМЕЧАНИЯ К ИСПОЛЬЗОВАНИЮ `SUPER()`

Помните следующие важные особенности, когда будете использовать `super()`:

- Функцию `super()` можно использовать только в конструкторе производного класса. Попытка вызвать ее в непроизводном классе (в классе, не содержащем `extends` в объявлении) или в функции вызовет ошибку.
- Функция `super()` должна вызываться перед любыми попытками использовать ссылку `this` в конструкторе. Функция `super()` отвечает за инициализацию `this`, поэтому попытка обратиться к `this` до вызова `super()` приведет к ошибке.
- Вызов `super()` можно опустить, только если конструктор класса возвращает объект.

Затенение методов класса

Методы в производных классах всегда затеняют одноименные методы базового класса. Например, в класс `Square` можно добавить метод `getArea()`, чтобы переопределить эту функцию:

```
class Square extends Rectangle {
    constructor(length) {
```

```
        super(length, length);
    }

    // переопределение и затенение Rectangle.prototype.getArea()
    getArea() {
        return this.length * this.length;
    }
}
```

Так как метод `getArea()` определяется как часть класса `Square`, метод `Rectangle.prototype.getArea()` становится недоступен экземплярам `Square`. Конечно, вы всегда можете вызвать версию метода из базового класса, используя метод `super.getArea()`, например:

```
class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }

    // переопределение и затенение Rectangle.prototype.getArea()
    getArea() {
        return super.getArea();
    }
}
```

Ссылка `super` в данном случае действует в точности как обсуждалось в главе 4 (раздел «Простой доступ к прототипу с помощью ссылки `super`»). Она автоматически корректирует ссылку `this` и позволяет просто вызвать метод.

Унаследованные статические члены

Если в базовом классе имеются статические члены, они также будут доступны в производном классе. Механизм наследования работает так же, как в других языках, но для JavaScript эта идея в новинку. Например:

```
class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }

    getArea() {
        return this.length * this.width;
    }

    static create(length, width) {
        return new Rectangle(length, width);
    }
}

class Square extends Rectangle {
    constructor length) {

        // эквивалент Rectangle.call(this, length, length)
    }
}
```

```
        super(length, length);
    }
}

var rect = Square.create(3, 4);

console.log(rect instanceof Rectangle); // true
console.log(rect.getArea());           // 12
console.log(rect instanceof Square);   // false
```

В этом фрагменте в класс `Rectangle` добавляется новый статический метод `create()`. Благодаря наследованию этот метод доступен также как `Square.create()` и действует точно так же, как метод `Rectangle.create()`.

Производные классы из выражений

Самой мощной особенностью производных классов в ECMAScript 6 является, пожалуй, возможность получать производные классы из выражений. После ключевого слова `extends` можно указать любое выражение, при условии, что оно возвращает функцию с методом `[[Construct]]` и прототипом. Например:

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
};

class Square extends Rectangle {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());           // 9
console.log(x instanceof Rectangle); // true
```

В этом примере определяется конструктор `Rectangle` в стиле ECMAScript 5 и класс `Square`. Поскольку `Rectangle` имеет `[[Construct]]` и прототип, класс `Square` все еще может наследовать тип `Rectangle` непосредственно.

Допустимость любых выражений после ключевого слова `extends` открывает широкие возможности, такие как динамическое определение унаследованного типа. Например:

```
function Rectangle(length, width) {
    this.length = length;
    this.width = width;
}

Rectangle.prototype.getArea = function() {
    return this.length * this.width;
}
```

```
};

function getBase() {
    return Rectangle;
}

class Square extends getBase() {
    constructor(length) {
        super(length, length);
    }
}

var x = new Square(3);
console.log(x.getArea());           // 9
console.log(x instanceof Rectangle); // true
```

Функция `getBase()` вызывается непосредственно в объявлении класса. Она возвращает `Rectangle`, что делает этот пример функционально эквивалентным предыдущему. А поскольку базовый класс может определяться динамически, появляется возможность создавать разные варианты наследования. Например, можно очень эффективно создавать классы-примеси, как показано ниже:

```
let SerializableMixin = {
    serialize() {
        return JSON.stringify(this);
    }
};

let AreaMixin = {
    getArea() {
        return this.length * this.width;
    }
}

function mixin(...mixins) {
    var base = function() {};
    Object.assign(base.prototype, ...mixins);
    return base;
}

class Square extends mixin(AreaMixin, SerializableMixin) {
    constructor(length) {
        super();
        this.length = length;
        this.width = length;
    }
}

var x = new Square(3);
console.log(x.getArea());           // 9
console.log(x.serialize());         // '{"length":3,"width":3}'
```

В этом примере вместо классического наследования используется прием смешивания. Функция `mixin()` принимает произвольное количество аргументов, представляющих смешиваемые объекты.

Она создает функцию с именем `base`, добавляет в прототип этой функции свойства, присутствующие в каждом подмешиваемом объекте, и возвращает функцию. Поэтому `Square` может воспользоваться ею в объявлении `extends`. Не забывайте, что при использовании `extends` требуется вызвать `super()` в конструкторе.

Экземпляр `Square` получает метод `getArea()` от `AreaMixin` и метод `serialize()` от `SerializableMixin`. Все это достигается за счет наследования прототипов. Функция `mixin()` динамически заполняет прототип новой функции собственными свойствами каждого подмешиваемого объекта. Имейте в виду, что если сразу несколько объектов будут иметь одноименное свойство, в прототипе функции останется только добавленное последним.

ПРИМЕЧАНИЕ

После ключевого слова `extends` можно использовать любое выражение, но не все выражения дают в результате допустимый класс. В частности, если указать значение `null` или функцию-генератор (рассматриваются в главе 8) после `extends`, это станет причиной ошибок. В этих случаях попытка создать новый экземпляр вызовет ошибку из-за невозможности вызвать `[[Construct]]`.

Наследование встроенных объектов

Практически с самого появления массивов в JavaScript разработчики пытаются создавать свои специализированные типы массивов через наследование. В ECMAScript 5 и более ранних версиях это было невозможно. Попытки использовать классическое наследование не давали в результате работающего кода. Например:

```
// поведение встроенного массива
var colors = [];
colors[0] = "red";
console.log(colors.length);    // 1

colors.length = 0;
console.log(colors[0]);        // undefined

// попытка унаследовать массив в ES5
function MyArray() {
    Array.apply(this, arguments);
}

MyArray.prototype = Object.create(Array.prototype, {
    constructor: {
        value: MyArray,
        writable: true,
        configurable: true,
        enumerable: true
    }
});

var colors = new MyArray();
colors[0] = "red";
console.log(colors.length);    // 0

colors.length = 0;
console.log(colors[0]);        // "red"
```


Вывод `console.log()` в конце примера показывает, как использование классической для JavaScript формы наследования массива приводит к неожиданному поведению. Свойство `length` и свойства с числовыми именами в экземпляре `MyArray` действуют совсем не так, как во встроенных массивах, потому что эта функциональность не может быть приобретена ни за счет вызова `Array.apply()`, ни за счет присваивания прототипа.

Одной из целей классов в ECMAScript 6 была возможность наследования всех встроенных объектов. Для ее достижения модель наследования классов была несколько изменена по сравнению с классической моделью наследования, использовавшейся в ECMAScript 5 и раньше, и получила два существенных отличия.

В классическом наследовании в стиле ECMAScript 5 производным типом (например, `MyArray`) создается значение ссылки `this`, а затем вызывается конструктор базового типа (например, `Array.apply()`). Это означает, что `this` начинает вести себя как экземпляр `MyArray` и затем снабжается дополнительными свойствами из `Array`.

В ECMAScript 6, напротив, в наследовании классов значение `this` создается базовым типом (`Array`) и затем модифицируется конструктором производного класса (`MyArray`). В результате `this` изначально получает всю функциональность базового типа и связанные с ней возможности.

Следующий пример демонстрирует, как действует специальный тип массивов, реализованный с помощью наследования классов:

```
class MyArray extends Array {
  // пустой
}

var colors = new MyArray();
colors[0] = "red";
console.log(colors.length);    // 1

colors.length = 0;
console.log(colors[0]);        // undefined
```

`MyArray` наследует встроенный тип `Array` и поэтому действует как `Array`. Операции со свойствами, имеющими числовые имена, оказывают влияние на свойство `length`, а манипуляции со свойством `length` отражаются на свойствах с числовыми именами. Это означает, что теперь вы можете унаследовать возможности `Array`, чтобы создать собственный производный класс массивов, а также унаследовать возможности и любых других встроенных объектов. Что касается этой дополнительной функциональности, то из ECMAScript 6 и производных классов фактически был убран последний особый случай наследования встроенных объектов, но в нем осталось еще кое-что, достойное исследования.

Свойство `Symbol.species`

Наследование встроенных объектов обладает одной интересной особенностью: любой метод, возвращающий экземпляр встроенного объекта, в производном классе автоматически будет возвращать экземпляр этого производного класса. Это означает, что если имеется производный класс `MyArray`, наследующий `Array`, его методы, такие как `slice()`, будут возвращать экземпляры `MyArray`. Например:

```
class MyArray extends Array {
  // пустой
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);
```

```
console.log(items instanceof MyArray);    // true
console.log(subitems instanceof MyArray);  // true
```

В этом фрагменте вызов метода `slice()` возвращает экземпляр `MyArray`. В действительности, метод `slice()`, унаследованный от объекта `Array`, возвращает экземпляр `Array`. Но за кулисами в игру вступает свойство `Symbol.species`, которое производит дополнительные изменения.

Стандартный символ `Symbol.species` используется для определения статического свойства с методом доступа, который возвращает функцию. Эта функция является конструктором, который применяется всякий раз, когда внутри метода экземпляра создается новый экземпляр класса (взамен использования конструктора). Свойство `Symbol.species` определено в следующих встроенных типах:

- `Array`;
- `ArrayBuffer` (обсуждается в главе 10);
- `Map`;
- `Promise`;
- `RegExp`;
- `Set`;
- типизированные массивы (обсуждаются в главе 10).

Все перечисленные типы по умолчанию имеют свойство `Symbol.species`, возвращающее `this`, то есть свойство всегда возвращает функцию-конструктор. Подобную функциональность можно реализовать в собственном классе, как показано ниже:

```
// некоторые встроенные типы используют species,
// как показано ниже
class MyClass {
  static get [Symbol.species]() {
    return this;
  }

  constructor(value) {
    this.value = value;
  }

  clone() {
    return new this.constructor[Symbol.species](this.value);
  }
}
```

В этом примере стандартный символ `Symbol.species` используется для создания в классе `MyClass` статического свойства с методом доступа. Обратите внимание, что определяется только метод чтения. Отсутствие метода записи объясняется невозможностью изменения специализации самого класса. Любой вызов `this.constructor[Symbol.species]` возвращает `MyClass`. Чтобы вернуть новый экземпляр, метод `clone()` использует это определение вместо прямого вызова `MyClass`. Данный прием позволяет производным классам переопределить возвращаемое значение. Например:

```
class MyClass {
  static get [Symbol.species]() {
    return this;
  }
  constructor(value) {
    this.value = value;
  }

  clone() {
    return new this.constructor[Symbol.species](this.value);
  }
}

class MyDerivedClass1 extends MyClass {
  // пустой
}

class MyDerivedClass2 extends MyClass {
  static get [Symbol.species]() {
    return MyClass;
  }
}

let instance1 = new MyDerivedClass1("foo"),
    clone1 = instance1.clone(),
    instance2 = new MyDerivedClass2("bar"),
    clone2 = instance2.clone();

console.log(clone1 instanceof MyClass);           // true
console.log(clone1 instanceof MyDerivedClass1);    // true
console.log(clone2 instanceof MyClass);           // true
console.log(clone2 instanceof MyDerivedClass2);    // false
```

Здесь `MyDerivedClass1` наследует класс `MyClass` и не переопределяет свойство `Symbol.species`. Когда вызывается метод `clone()`, он возвращает экземпляр `MyDerivedClass1`, потому что `this.constructor[Symbol.species]` возвращает `MyDerivedClass1`. Класс `MyDerivedClass2` также наследует `MyClass` и переопределяет свойство `Symbol.species` так, что его метод чтения возвращает `MyClass`. Когда вызывается метод `clone()` экземпляра `MyDerivedClass2`, возвращаемое им значение оказывается экземпляром `MyClass`. С помощью `Symbol.species` любой производный класс может определить, какой тип должен иметь экземпляр, возвращаемый методом.

Например, `Array` использует `Symbol.species`, чтобы определить класс, который должен применяться методами, возвращающими массив. В классе, производном от `Array`, вы можете определить тип объекта, возвращаемого унаследованными методами, например:

```
class MyArray extends Array {
  static get [Symbol.species]() {
    return Array;
  }
}

let items = new MyArray(1, 2, 3, 4),
    subitems = items.slice(1, 3);
```

```
console.log(items instanceof MyArray);      // true
console.log(subitems instanceof Array);     // true
console.log(subitems instanceof MyArray);   // false
```

Этот фрагмент переопределяет свойство `Symbol.species` в `MyArray`, унаследованное от `Array`. Все унаследованные методы, возвращающие массивы, теперь будут создавать экземпляры `Array` вместо `MyArray`.

Вообще, всякий раз когда в методе класса возникает необходимость вызвать `this.constructor`, вместо него следует применять свойство `Symbol.species`. Это позволит производным классам без лишних хлопот переопределить тип возвращаемого значения. Кроме того, создавая классы, производные от класса, в котором определено свойство `Symbol.species`, используйте это значение вместо конструктора.

Использование `new.target` в конструкторах классов

В главе 3 вы познакомились с метасвойством `new.target` и узнали, как изменяется его значение в зависимости от способа вызова функции. Метасвойство `new.target` можно также использовать в конструкторах классов, чтобы определить, как вызывался класс. В простейшем случае `new.target` содержит ссылку на функцию-конструктор класса, как в следующем примере:

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    this.length = length;
    this.width = width;
  }
}

// new.target - это Rectangle
var obj = new Rectangle(3, 4); // выведет true
```

Этот пример показывает, что в вызове `new Rectangle(3, 4)` метасвойство `new.target` ссылается на `Rectangle`. Конструкторы классов не могут вызываться без ключевого слова `new`, поэтому в конструкторах классов метасвойство `new.target` всегда определено. Но его значение не всегда будет тем же самым. Взгляните на следующий фрагмент:

```
class Rectangle {
  constructor(length, width) {
    console.log(new.target === Rectangle);
    this.length = length;
    this.width = width;
  }
}

class Square extends Rectangle {
  constructor(length) {
    super(length, length)
  }
}

// new.target - это Square
var obj = new Square(3); // выведет false
```

Здесь конструктор `Rectangle` вызывается конструктором `Square`, поэтому в конструкторе `Rectangle` метасвойство `new.target` будет ссылаться на `Square`. Это важно потому, что дает возможность менять поведение конструктора в зависимости от того, как он был вызван. Например, с помощью `new.target` можно создать абстрактный базовый класс (экземпляры которого нельзя создавать непосредственно), как показано ниже:

```
// абстрактный базовый класс
class Shape {
  constructor() {
    if (new.target === Shape) {
      throw new Error("This class cannot be instantiated directly.")
    }
  }
}

class Rectangle extends Shape {
  constructor(length, width) {
    super();
    this.length = length;
    this.width = width;
  }
}

var x = new Shape();           // вызовет ошибку
var y = new Rectangle(3, 4);   // нет ошибки
console.log(y instanceof Shape); // true
```

В этом примере конструктор класса `Shape` возбуждает ошибку, если `new.target` ссылается на `Shape`, то есть вызов `new Shape()` всегда будет вызывать ошибку. Однако `Shape` можно использовать в роли базового класса, что и делает класс `Rectangle`. Вызов `super()` выполняет конструктор `Shape`, а так как в этом случае `new.target` ссылается на `Rectangle`, этот конструктор продолжает работу, не возбуждая ошибки.

ПРИМЕЧАНИЕ

Классы не могут вызываться без ключевого слова `new`, поэтому внутри конструкторов классов метасвойство `new.target` никогда не будет иметь значения `undefined`.

В заключение

Классы, появившиеся в ECMAScript 6, упрощают реализацию наследования в JavaScript, и вам пригодится опыт применения механизмов наследования в других языках, если он у вас имеется. Первоначально классы в ECMAScript 6 играли роль синтаксического сахара для классической модели наследования в ECMAScript 5, но постепенно в них были добавлены дополнительные особенности, уменьшающие вероятность ошибок.

Классы в ECMAScript 6 используют механизм наследования прототипов для определения статических методов в прототипах классов, тогда как статические методы добавляются непосредственно в конструктор. Все методы создаются как неперечислимые, что более точно соответствует поведению встроенных объектов, чьи методы обычно являются неперечислимыми по умолчанию. Кроме того, конструкторы классов не могут вызываться без ключевого слова `new`, что гарантирует невозможность вызова класса как обычной функции по ошибке.

Наследование на основе классов позволяет создавать классы, производные от других классов, функций или выражений. Таким образом, вызовом функции можно определить базу для наследования, что дает возможность создавать примеси и применять другие шаблоны комбинирования с целью создания нового класса. Механизм наследования теперь действует так, что позволяет наследовать встроенные объекты, такие как `Array`, и получать в результате ожидаемое поведение.

Вы можете использовать метасвойство `new.target` в конструкторах классов, чтобы определять разные варианты поведения в зависимости от того, как класс вызывался. Чаще всего эта возможность используется для определения абстрактных базовых классов, которые возбуждают ошибку при попытке создать их экземпляры непосредственно, но допускают наследование другими классами.

В целом классы являются важным дополнением языка JavaScript. Они обеспечивают более компактный синтаксис и более широкие возможности для определения пользовательских типов объектов безопасным и непротиворечивым способом.

Расширенные возможности массивов

Массив — это основополагающий объект в JavaScript. Но если другие аспекты JavaScript продолжали развиваться с течением времени, массивы пребывали в неизменном состоянии, пока спецификация ECMAScript 5 не добавила несколько дополнительных методов, упрощающих их применение. ECMAScript 6 продолжила совершенствовать работу с массивами и добавила еще несколько возможностей, в том числе новые методы создания, несколько удобных вспомогательных методов и поддержку типизированных массивов. В данной главе мы последовательно рассмотрим все эти нововведения.

Создание массивов

До ECMAScript 6 существовало два основных способа создания массивов: с помощью конструктора `Array` и с использованием синтаксиса литералов массивов. Оба подхода требовали перечислить элементы будущего массива по отдельности и имели существенные ограничения. Возможности преобразования объектов, подобных массивам (то есть объектов с числовыми индексами и свойством `length`), в массивы также имели ограничения и часто требовали дополнительного кода. Чтобы упростить создание массивов JavaScript, в ECMAScript 6 были добавлены методы `Array.of()` и `Array.from()`.

Метод `Array.of()`

Одной из причин добавления новых методов в JavaScript было стремление избавить разработчиков от ошибок при создании массивов с помощью конструктора `Array`, поскольку этот конструктор проявляет разное поведение в зависимости от типов и количества аргументов. Например:

```
let items = new Array(2);
console.log(items.length); // 2
console.log(items[0]);     // undefined
console.log(items[1]);     // undefined

items = new Array("2");
console.log(items.length); // 1
console.log(items[0]);     // "2"

items = new Array(1, 2);
console.log(items.length); // 2
```

```
console.log(items[0]);      // 1
console.log(items[1]);      // 2

items = new Array(3, "2");
console.log(items.length);  // 2
console.log(items[0]);      // 3
console.log(items[1]);      // "2"
```

Когда конструктор `Array` получает единственное числовое значение, он присваивает это значение свойству `length` вновь созданного массива. Если передать ему единственное нечисловое значение, это значение становится единственным элементом нового массива. Если передать несколько значений (числовых или нечисловых), эти значения становятся элементами нового массива. Это довольно запутанное и рискованное поведение, потому что тип передаваемых данных известен далеко не всегда.

Чтобы решить эту проблему, спецификация ECMAScript 6 добавила метод `Array.of()`. Он действует подобно конструктору `Array`, но не рассматривает единственный числовой аргумент как специальный случай. Метод `Array.of()` всегда создает массив, содержащий аргументы метода независимо от их количества или типов. Ниже приводится несколько примеров использования метода `Array.of()`:

```
let items = Array.of(1, 2);
console.log(items.length);  // 2
console.log(items[0]);      // 1
console.log(items[1]);      // 2

items = Array.of(2);
console.log(items.length);  // 1
console.log(items[0]);      // 2

items = Array.of("2");
console.log(items.length);  // 1
console.log(items[0]);      // "2"
```

Чтобы создать массив с помощью метода `Array.of()`, достаточно передать ему значения, которые требуется включить в массив. Первый из примеров выше создает массив с двумя числами, второй — массив с одним числом и последний — массив с одной строкой. Этот подход напоминает использование литерала массива, и в большинстве случаев для создания простых массивов вместо метода `Array.of()` можно использовать литералы массивов. Но если понадобится передать конструктор `Array` в функцию, возможно, вы предпочтете вместо него передать `Array.of()`, чтобы гарантировать непротиворечивое поведение. Например:

```
function createArray(arrayCreator, value) {
    return arrayCreator(value);
}

let items = createArray(Array.of, value);
```

Здесь функция `createArray()` принимает функцию создания массива и значение для добавления в массив. Вы можете передать ей `Array.of()` в первом аргументе, чтобы создать новый массив. Было бы опасно передавать такой функции конструктор `Array` непосредственно, если нет гарантии, что `value` может принимать только нечисловые значения.

ПРИМЕЧАНИЕ

Метод `Array.of()` не использует свойство `Symbol.species` (см. раздел «Свойство `Symbol.species`» в главе 9) для определения типа возвращаемого значения. Вместо этого используется текущий конструктор (`this` внутри метода `of()`).

Метод `Array.from()`

Преобразование объектов, не являющихся массивами, в настоящие массивы всегда было сложной задачей в JavaScript. Например, если вы пожелаете использовать объект `arguments` (объект, подобный массиву) как массив, вам придется сначала преобразовать его в массив. Для преобразования в массивы объектов, подобных массивам, в ECMAScript 5 требовалось написать функцию, например, как показано ниже:

```
function makeArray(arrayLike) {
    var result = [];

    for (var i = 0, len = arrayLike.length; i < len; i++) {
        result.push(arrayLike[i]);
    }

    return result;
}

function doSomething() {
    var args = makeArray(arguments);

    // использовать args
}
```

Здесь вручную создается массив `result`, и каждый элемент из `arguments` копируется в новый массив. Это вполне работоспособное решение, но оно требует написания дополнительного кода для относительно простой операции. Со временем разработчики открыли способ, позволяющий уменьшить объем кода, если применить встроенный метод `slice()` к объекту, подобному массиву, например:

```
function makeArray(arrayLike) {
    return Array.prototype.slice.call(arrayLike);
}

function doSomething() {
    var args = makeArray(arguments);

    // использовать args
}
```

Этот код функционально эквивалентен предыдущему примеру и в своей работе опирается на передачу в метод `slice()` ссылки `this`, указывающей на объект, подобный массиву. Так как методу `slice()` для работы необходимы только свойства с числовыми именами и свойство `length`, он прекрасно справляется с любыми объектами, подобными массивам.

Но даже при том, что это решение не требует вводить много программного кода, вызов `Array.prototype.slice.call(arrayLike)` явно не выглядит как команда «преобразовать `arrayLike` в массив». К счастью, ECMAScript 6 добавила метод `Array.from()` как очевидное средство преобразования объектов в массивы.

Принимая в первом аргументе итерируемый объект или объект, подобный массиву, метод `Array.from()` возвращает массив, как показано в следующем простом примере:

```
function doSomething() {
    var args = Array.from(arguments);

    // использовать args
}
```

Вызов `Array.from()` создает новый массив из элементов в `arguments`. Таким образом, `args` — это экземпляр `Array`, содержащий те же значения и в тех же позициях, что и `arguments`.

ПРИМЕЧАНИЕ

Для определения типа возвращаемого массива метод `Array.from()` также использует `this`.

Преобразование с отображением

Если необходимо выполнить более сложное преобразование, методу `Array.from()` во втором аргументе можно передать функцию отображения. Эта функция должна принимать каждое значение из объекта, подобного массиву, и преобразовывать его в некоторую окончательную форму перед сохранением в соответствующей позиции в массиве. Взгляните на следующий пример:

```
function translate() {
    return Array.from(arguments, (value) => value + 1);
}
```

```
let numbers = translate(1, 2, 3);
console.log(numbers);    // 2,3,4
```

Здесь в `Array.from()` передается `(value) => value + 1` в качестве функции отображения, то есть она прибавляет 1 к каждому элементу перед его сохранением в массиве. Если функция отображения является методом объекта, в третьем необязательном аргументе методу `Array.from()` можно передать значение для ссылки `this` внутри функции отображения:

```
let helper = {
    diff: 1,

    add(value) {
        return value + this.diff;
    }
};

function translate() {
    return Array.from(arguments, helper.add, helper);
}

let numbers = translate(1, 2, 3);

console.log(numbers);    // 2,3,4
```

В этом примере роль функции отображения играет метод `helper.add()`. Так как `helper.add()` использует свойство `this.diff`, в вызов `Array.from()` требуется передать третий аргумент, определяющий значение `this`. Благодаря третьему аргументу `Array.from()` легко может преобразовывать данные, не вызывая `bind()` и не определяя значение `this` каким-либо другим способом.

Использование итерируемых объектов

Метод `Array.from()` работает с итерируемыми объектами и объектами, подобными массивам. Это означает, что метод может преобразовывать в массивы любые объекты, обладающие свойством `Symbol.iterator`. Например:

```
let numbers = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  }
};

let numbers2 = Array.from(numbers, (value) => value + 1);

console.log(numbers2);    // 2,3,4
```

Так как объект `numbers` является итерируемым, его можно передать в вызов метода `Array.from()` и преобразовать в массив. Функция отображения прибавляет 1 к каждому числу, поэтому получившийся массив содержит 2, 3 и 4 вместо 1, 2 и 3.

ПРИМЕЧАНИЕ

Если объект является одновременно итерируемым и подобным массиву, для получения значений `Array.from()` использует итератор.

Новые методы всех массивов

Продолжая курс, взятый спецификацией ECMAScript 5, ECMAScript 6 добавила в массивы несколько новых методов. Методы `find()` и `findIndex()` добавлены с целью помочь разработчикам использовать массивы с любыми значениями, тогда как идея добавления методов `fill()` и `copyWithin()` была навеяна некоторыми случаями использования *типизированных массивов*. Типизированные массивы, появившиеся в ECMAScript 6, — это разновидность массивов, которые могут хранить только числовые значения.

Методы `find()` и `findIndex()`

До ECMAScript 5 поиск в массивах осложнялся из-за отсутствия встроенных методов, позволяющих это. ECMAScript 5 добавила методы `indexOf()` и `lastIndexOf()`, которые наконец дали разработчикам возможность искать конкретные значения в массивах. Эти два метода были значительным усовершенствованием, но они весьма ограничены в своих возможностях, позволяя искать значения только по одному. Например, если понадобится найти первое четное число в последовательности, вам придется написать для этого свой код. ECMAScript 6 решила эту проблему, добавив методы `find()` и `findIndex()`.

Оба метода, `find()` и `findIndex()`, принимают два аргумента: функцию обратного вызова и необязательное значение для `this` внутри нее. Функции обратного вызова передается элемент массива, индекс этого элемента в массиве и ссылка на массив — те же аргументы, что передаются в такие методы, как `map()` и `forEach()`. Функция должна вернуть `true`, если данное значение соответствует вашим критериям поиска. Оба метода, `find()` и `findIndex()`, прекращают поиск, как только функция вернет `true`.

Единственное отличие между этими двумя методами состоит в том, что `find()` возвращает значение, а `findIndex()` — индекс найденного значения. Эти отличия демонстрирует следующий пример:

```
let numbers = [25, 30, 35, 40, 45];

console.log(numbers.find(n => n > 33));      // 35
console.log(numbers.findIndex(n => n > 33));  // 2
```

Этот код вызывает `find()` и `findIndex()`, чтобы найти в массиве `numbers` первое значение, большее 33. Вызов `find()` возвращает 35, а вызов `findIndex()` возвращает 2 — индекс числа 35 в массиве `numbers`.

Оба метода, `find()` и `findIndex()`, удобно использовать для поиска в массивах элементов, соответствующих условиям, а не определенному значению. Если требуется просто найти значение, лучшим выбором будут методы `indexOf()` и `lastIndexOf()`.

Метод `fill()`

Метод `fill()` заполняет один или несколько элементов массива определенным значением. Когда передается одно значение, `fill()` затирает им все значения в массиве. Например:

```
let numbers = [1, 2, 3, 4];

numbers.fill(1);

console.log(numbers.toString());  // 1,1,1,1
```

Здесь вызов `numbers.fill(1)` приводит к записи числа 1 во все элементы в массиве `numbers`. Если необходимо изменить лишь несколько элементов, дополнительно можно передать начальный и конечный индексы группы элементов для замены, при этом элемент с конечным индексом не будет изменен, например:

```
let numbers = [1, 2, 3, 4];

numbers.fill(1, 2);

console.log(numbers.toString());  // 1,2,1,1

numbers.fill(0, 1, 3);

console.log(numbers.toString());  // 1,0,0,1
```

Аргумент 2 в вызове `numbers.fill(1, 2)` требует начать заполнение с элемента с индексом 2. Третий аргумент с конечным индексом не указан, поэтому его роль возьмет на себя значение `numbers.length`, то есть число 4 будет записано в два последних элемента. Операция `numbers.fill(0, 1, 3)` запишет 0 в элементы массива с индексами 1 и 2. Вызов `fill()` со вторым и третьим аргументами позволяет заполнить сразу несколько указанных элементов, не затирая остальные элементы массива.

ПРИМЕЧАНИЕ

Если в начальном или конечном индексе передать отрицательное число, это число будет прибавляться к значению длины массива, чтобы получить искомое местоположение. Например, значение `-1` в аргументе, определяющем начальный индекс, соответствует индексу `array.length - 1`, где `array` — это массив, для которого был вызван метод `fill()`.

Метод `copyWithin()`

Метод `copyWithin()` напоминает `fill()` тем, что также изменяет сразу несколько элементов массива. Однако вместо единственного значения для записи в элементы массива `copyWithin()` позволяет копировать элементы в массиве. Для этого методу `copyWithin()` нужно передать два аргумента: индекс первого элемента в массиве, куда должно выполняться копирование, и индекс первого элемента в массиве, откуда должны извлекаться значения.

Например, скопировать значения первых двух элементов в массиве в последние два элемента в этом же массиве можно так:

```
let numbers = [1, 2, 3, 4];

// вставить значения в массив, начиная с индекса 2,
// копировать значения из массива, начиная с индекса 0
numbers.copyWithin(2, 0);

console.log(numbers.toString()); // 1,2,1,2
```

Этот код вставит значения в массив `numbers`, начиная с индекса 2, то есть будут затерты элементы с индексами 2 и 3. Значение 0 во втором аргументе сообщает методу `copyWithin()`, что значения для копирования должны извлекаться из элементов, начиная с индекса 0, и копирование должно продолжаться, пока не будет записан последний элемент массива.

По умолчанию `copyWithin()` всегда копирует значения до конца массива, но вы можете передать необязательный третий аргумент и ограничить количество изменяемых элементов. Этот третий аргумент представляет индекс элемента, по достижении которого копирование должно быть прекращено, при этом элемент с этим индексом не будет скопирован. Ниже показано, как такая операция выглядит в коде:

```
let numbers = [1, 2, 3, 4];

// вставить значения в массив, начиная с индекса 2,
// копировать значения из массива, начиная с индекса 0
// copy values from array starting at index 0
// остановить копирование по достижении индекса 1
numbers.copyWithin(2, 0, 1);

console.log(numbers.toString()); // 1,2,1,4
```

Этот пример скопирует только элемент с индексом 0, потому что в третьем необязательном аргументе передан индекс 1 элемента, который уже не должен копироваться. В результате последний элемент массива не изменился.

ПРИМЕЧАНИЕ

Так же как при использовании метода `fill()`, если в любом аргументе передать методу `copyWithin()` отрицательное число, это число будет прибавляться к значению длины массива, чтобы получить искомый индекс.

Полезность методов `fill()` и `copyWithin()` в данный момент может показаться вам неочевидной. Это объясняется тем, что первоначально они создавались для типизированных массивов и были добавлены в обычные массивы только ради единообразия. Однако, как вы узнаете из следующего раздела, эти методы оказываются намного более ценными, если используются с типизированными массивами для управления последовательностями чисел

Типизированные массивы

Типизированные массивы — это особые массивы, предназначенные для работы с числовыми типами (не со всеми типами, как можно было бы предположить). Своими корнями типизированные массивы уходят в WebGL — реализацию OpenGL ES 2.0 для использования в веб-страницах с элементом `<canvas>`. Типизированные массивы были созданы как часть этой реализации для поддержки быстрой поразрядной арифметики в JavaScript.

Арифметические операции со встроенными числами JavaScript выполняются слишком медленно для WebGL, потому что числа хранятся в 64-разрядном вещественном формате и при необходимости преобразуются в 32-разрядные целые числа. Типизированные массивы создавались с целью обойти это ограничение и обеспечить более высокую производительность арифметических операций. Идея состоит в том, что любое отдельное число можно интерпретировать как массив битов и таким образом использовать знакомые методы, доступные в массивах JavaScript.

ECMAScript 6 официально добавила поддержку типизированных массивов в язык, чтобы обеспечить лучшую совместимость движков JavaScript и функциональное сходство с массивами JavaScript. Несмотря на то что типизированные массивы в ECMAScript 6 не полностью соответствуют типизированным массивам в WebGL, они имеют достаточно много общего, чтобы считать версию ECMAScript 6 дальнейшим развитием версии WebGL, а не каким-то другим решением.

Числовые типы данных

Числа в JavaScript хранятся в формате IEEE 754, который используется для представления 64-разрядных вещественных чисел. Этот формат представляет целые и вещественные числа в JavaScript, и при изменении чисел часто происходят преобразования между двумя форматами. Типизированные массивы позволяют хранить и манипулировать восемью разными числовыми типами:

- 8-разрядные целые со знаком (`int8`);
- 8-разрядные целые без знака (`uint8`);
- 16-разрядные целые со знаком (`int16`);
- 16-разрядные целые без знака (`uint16`);
- 32-разрядные целые со знаком (`int32`);
- 32-разрядные целые без знака (`uint32`);
- 32-разрядные вещественные (`float32`);
- 64-разрядные вещественные (`float64`).

Если представить число, уместящееся в 8 двоичных разрядов, как обычное число JavaScript, напрасно будет потеряно 56 бит. Эти биты могли бы использоваться для хранения дополнительных 8-разрядных значений или другого числа, занимающего меньше 56 бит. Более эффективное использование памяти — одна из задач, решаемых типизированными массивами.

Все операции и объекты, связанные с типизированными массивами, опираются на эти восемь типов данных. Но чтобы использовать их для хранения данных, необходимо создать буфер массива.

ПРИМЕЧАНИЕ

Далее в книге для ссылки на эти восемь типов я буду использовать сокращения, указанные в круглых скобках. Эти сокращения не являются частью языка JavaScript; это просто более короткий способ отразить то, что потребовало бы более длинного описания.

Буферы массивов

Основой всех типизированных массивов является *буфер массива* — фрагмент памяти, способный хранить заданное количество байтов. Создание буфера массива сродни вызову функции `malloc()` в языке C, которая выделяет память, не определяя, что в ней будет храниться. Буфер массива можно создать с помощью конструктора `ArrayBuffer`:

```
let buffer = new ArrayBuffer(10); // выделит 10 байт
```

Просто передайте в вызов конструктора число байт, которые должен содержать буфер. Эта инструкция `let` создаст буфер массива длиной в 10 байт. После создания буфера можно узнать его размер в байтах с помощью свойства `byteLength`:

```
let buffer = new ArrayBuffer(10); // выделит 10 байт
console.log(buffer.byteLength); // 10
```

Новый буфер массива, содержащий часть существующего буфера, можно создать с помощью метода `slice()`. Метод `slice()` буфера действует точно так же, как метод `slice()` массивов: вы передаете ему начальный и конечный индексы в качестве аргументов, а он возвращает новый экземпляр `ArrayBuffer`, содержащий копии элементов из оригинала. Например:

```
let buffer = new ArrayBuffer(10); // выделит 10 байт
let buffer2 = buffer.slice(4, 6);
console.log(buffer2.byteLength); // 2
```

Этот фрагмент создаст новый буфер `buffer2` и скопирует в него байты с индексами 4 и 5. По аналогии с версией этого метода для массивов второй аргумент метода `slice()` представляет индекс, по достижении которого копирование прекращается, и сам элемент с этим индексом не копируется.

Конечно, было бы бессмысленно выделять память, если бы не было возможности записывать в нее данные. Однако для этого нужно создать представление.

ПРИМЕЧАНИЕ

Буфер массива всегда представляет блок памяти с размером в байтах, указанным в момент его создания. Вы можете изменять данные, содержащиеся в буфере, но размер буфера — никогда.

Управление буферами массивов с помощью представлений

Буферы массивов представляют фрагменты памяти, а *представления* служат интерфейсом, посредством которого осуществляется управление содержимым в этой памяти. Представление позволяет оперировать буфером массива или подмножеством байтов в нем, читать и записывать данные одного из числовых типов. Тип `DataView` — это обобщенное представление буфера массива, позволяющее оперировать всеми восемью числовыми типами данных.

Чтобы задействовать `DataView`, сначала нужно создать экземпляр `ArrayBuffer` и с его помощью создать новый экземпляр `DataView`. Например:

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer);
```

Объект `view` в этом примере имеет доступ ко всем 10 байтам в буфере. Поддерживается также возможность создания представления для части буфера. Просто укажите смещение в байтах от начала буфера и, что необязательно, количество представляемых байтов. Если количество байтов не указано, `DataView` будет представлять часть буфера от указанного смещения до конца буфера. Например:

```
let buffer = new ArrayBuffer(10),
    view = new DataView(buffer, 5, 2);    // охватывает байты 5 и 6
```

Здесь `view` представляет только байты с индексами 5 и 6. Такой подход дает возможность создать несколько представлений в одном буфере, что может пригодиться, когда для всего приложения желательно использовать один блок памяти, а не выделять память динамически по мере необходимости.

Извлечение информации о представлении

Информацию о представлении можно получить с помощью следующих свойств, доступных только для чтения:

- `buffer`. Массив буфера, с которым связано представление.
- `byteOffset`. Второй аргумент в вызове конструктора `DataView`, если указан (по умолчанию 0).
- `byteLength`. Третий аргумент в вызове конструктора `DataView`, если указан (по умолчанию получает значение свойства `byteLength` буфера).

С помощью этих свойств можно точно определить, какой части буфера соответствует представление, например:

```
let buffer = new ArrayBuffer(10),
    view1 = new DataView(buffer),           // охватывает все байты
    view2 = new DataView(buffer, 5, 2);     // охватывает байты 5 и 6

console.log(view1.buffer === buffer);      // true
console.log(view2.buffer === buffer);      // true
console.log(view1.byteOffset);              // 0
console.log(view2.byteOffset);              // 5
console.log(view1.byteLength);              // 10
console.log(view2.byteLength);              // 2
```

Этот код создаст `view1`, представление для всего буфера, и `view2`, представление для малой части буфера. Эти представления будут иметь одинаковые значения свойств `buffer`, потому что оба связаны с одним и тем же буфером. Однако свойства `byteOffset` и `byteLength` этих представлений будут отличаться. Они отражают части буфера, на которые распространяется действие каждого представления.

Конечно, информация о представляемой области памяти сама по себе не особенно полезна. Необходимо еще иметь возможность читать данные из памяти и записывать их в нее.

Чтение и запись данных

Для каждого из восьми числовых типов данных в прототипе `DataView` имеется метод для записи и метод для чтения данных из буфера массива. Имена всех методов начинаются со слова *set* или *get*, за которым следует аббревиатура типа данных. Например, ниже перечисляются методы чтения и записи для операций со значениями `int8` и `uint8`:

- `getInt8(byteOffset, littleEndian)`. Читает значение `int8`, начиная с `byteOffset`.
- `setInt8(byteOffset, value, littleEndian)`. Записывает значение `int8`, начиная с `byteOffset`.

- `getUint8(byteOffset, littleEndian)`. Читает значение `uint8`, начиная с `byteOffset`.
- `setUint8(byteOffset, value, littleEndian)`. Записывает значение `uint8`, начиная с `byteOffset`.

Get-методы принимают два аргумента: смещение в байтах, откуда начинать чтение, и необязательное логическое значение, сообщающее, должно ли значение читаться как число с обратным порядком следования байтов. (Под *обратным порядком следования байтов* (*little-endian*) подразумевается такой порядок, когда младший значащий байт хранится в памяти первым, а не последним.) Set-методы принимают три аргумента: смещение в байтах, откуда начинать запись, значение для записи и необязательное логическое значение, сообщающее, должно ли значение записываться в формате с обратным порядком следования байтов.

В списке выше перечислены только методы, которые можно использовать для работы с 8-разрядными значениями, однако аналогичные методы существуют для работы с 16- и 32-разрядными значениями. Просто замените 8 в имени требуемого метода на 16 или 32. Помимо методов для работы с целочисленными значениями `DataView` имеет также следующие методы для чтения и записи вещественных чисел:

- `getFloat32(byteOffset, littleEndian)`. Читает значение `float32`, начиная с `byteOffset`.
- `setFloat32(byteOffset, value, littleEndian)`. Записывает значение `float32`, начиная с `byteOffset`.
- `getFloat64(byteOffset, littleEndian)`. Читает значение `float64`, начиная с `byteOffset`.
- `setFloat64(byteOffset, value, littleEndian)`. Записывает значение `float64`, начиная с `byteOffset`.

Следующий пример демонстрирует применение методов чтения и записи:

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
view.setInt8(1, -1);

console.log(view.getInt8(0)); // 5
console.log(view.getInt8(1)); // -1
```

В этом примере двухбайтный буфер используется для хранения двух значений `int8`. Первое значение сохраняется со смещением 0, а второе — со смещением 1, чем подтверждается, что каждое значение занимает ровно один байт (8 бит). Затем эти значения извлекаются из своих позиций вызовами метода `getInt8()`. В этом примере используются значения `int8`, но аналогичные операции можно производить с любыми из восьми числовых типов данных, используя соответствующие методы.

Представления имеют уникальную особенность, позволяя читать и записывать данные в любых форматах и в любой момент времени независимо от того, как эти данные сохранялись перед этим. Например, можно записать два значения `int8` и прочитать их как одно значение `int16`:

```
let buffer = new ArrayBuffer(2),
    view = new DataView(buffer);

view.setInt8(0, 5);
```

```
view.setInt8(1, -1);

console.log(view.getInt16(0)); // 1535
console.log(view.getInt8(0)); // 5
console.log(view.getInt8(1)); // -1
```

Вызов `view.getInt16(0)` прочитает все байты в представлении и интерпретирует их как число 1535. Чтобы понять суть происходящего, взгляните на рис. 10.1, где показано, как каждый вызов `setInt8()` изменяет содержимое буфера.

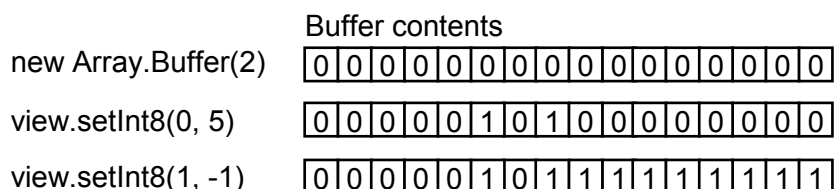


Рис. 10.1. Буфер массива после двух вызовов метода

Изначально буфер содержит 16 нулевых битов. Первый вызов `setInt8()`, записывающий число 5 в первый байт, изменяет значение двух битов на 1 (в 8-разрядном представлении число 5 имеет вид 00000101). Запись числа — 1 во второй байт изменяет все биты в этом байте на 1, что является двоичным представлением числа — 1. После второго вызова `setInt8()` буфер содержит 16 бит, и `getInt16()` прочитал эти биты как одно 16-разрядное целое число со знаком, которое в десятичном представлении имеет вид 1535.

Объект `DataView` прекрасно подходит для случаев, когда приходится смешивать типы данных подобным способом. Однако если вы используете только один определенный тип данных, лучшим выбором будет применение специализированных представлений.

Типизированные массивы — это представления

Типизированные массивы в ECMAScript 6 фактически являются специализированными представлениями буферов массивов. Вместо обобщенного объекта `DataView` для работы с буфером можно использовать объекты, специализированные для определенных типов данных. Восемью числовым типам данных соответствуют восемь специализированных представлений плюс один дополнительный вариант для значений `uint8`. В табл. 10.1 приводится сокращенная версия списка специализированных представлений из раздела 22.2 спецификации ECMAScript 6.

Таблица 10.1. Некоторые из специализированных представлений в ECMAScript 6

Имя конструктора	Размер элемента (в байтах)	Описание	Эквивалентный тип в языке C
<code>Int8Array</code>	1	8-разрядное целое число со знаком	<code>signed char</code>
<code>Uint8Array</code>	1	8-разрядное целое число без знака	<code>unsigned char</code>
<code>Uint8ClampedArray</code>	1	8-разрядное целое число без знака (с ограничивающим преобразованием)	<code>unsigned char</code>
<code>Int16Array</code>	2	16-разрядное целое число со знаком	<code>short</code>
<code>Uint16Array</code>	2	16-разрядное целое число без знака	<code>unsigned short</code>
<code>Int32Array</code>	4	32-разрядное целое число со знаком	<code>int</code>
<code>Uint32Array</code>	4	32-разрядное целое число без знака	<code>int</code>
<code>Float32Array</code>	4	32-разрядное вещественное число в формате IEEE	<code>float</code>
<code>Float64Array</code>	8	64-разрядное вещественное число в формате IEEE	<code>double</code>

В столбце «Имя конструктора» перечислены конструкторы типизированных массивов, а в других столбцах кратко описываются данные, которые могут храниться в этих массивах. Тип `Uint8ClampedArray` отличается от типа `Uint8Array` лишь тем, что значения в буфере не могут быть меньше 0 или больше 255. Значения, которые меньше 0, `Uint8ClampedArray` преобразует в 0 (—1, например, будет преобразовано в 0), а значения, которые больше 255, — в 255 (то есть число 300 превратится в 255).

Операции с типизированными массивами работают только с данными конкретного типа. Например, все операции с `Int8Array` выполняются над значениями `int8`. Размер элемента в типизированном массиве также зависит от его типа. Так, элемент массива `Int8Array` занимает всего один байт, а элемент `Float64Array` — восемь байтов. К счастью, обращаться к элементам можно по числовым индексам, как и в обычных массивах, что позволяет избежать некоторых не очень удобных вызовов `set`- и `get`-методов из `DataView`.

Создание специализированных представлений

Конструкторы типизированных массивов принимают аргументы разных типов, поэтому типизированные массивы можно создавать несколькими способами. Во-первых, новый типизированный массив можно создать, передав конструктору те же аргументы, которые принимает `DataView` (буфер массива, необязательное смещение в байтах и необязательная длина в байтах). Например:

```
let buffer = new ArrayBuffer(10),
    view1 = new Int8Array(buffer),
    view2 = new Int8Array(buffer, 5, 2);

console.log(view1.buffer === buffer); // true
console.log(view2.buffer === buffer); // true
console.log(view1.byteOffset);        // 0
console.log(view2.byteOffset);        // 5
console.log(view1.byteLength);        // 10
console.log(view2.byteLength);        // 2
```

В этом примере буфер `buffer` используется двумя представлениями экземпляра `Int8Array`. Оба представления, `view1` и `view2`, имеют те же свойства — `buffer`, `byteOffset` и `byteLength`, — что и экземпляры `DataView`. Поэтому, если используется только один числовой тип, вы легко сможете перейти на типизированные массивы везде, где используется `DataView`.

Второй способ создания типизированных массивов — передача конструктору единственного числа. Это число представляет количество элементов (не байтов) в массиве. Конструктор создаст новый буфер требуемого объема для хранения указанного количества элементов, и вы можете узнать, сколько элементов хранится в массиве, воспользовавшись свойством `length`. Например:

```
let ints = new Int16Array(2),
    floats = new Float32Array(5);
console.log(ints.byteLength); // 4
console.log(ints.length);    // 2

console.log(floats.byteLength); // 20
console.log(floats.length);    // 5
```

Здесь создается массив `ints` размером в два элемента. Каждое 16-разрядное целое число занимает два байта, поэтому всего массив занимает четыре байта. Массив `floats` имеет размер в пять элементов, поэтому для его размещения требуется 20 байтов (по четыре байта на элемент). В обоих случаях создается новый буфер, доступный через свойство `buffer`.

ПРИМЕЧАНИЕ

Если конструктор типизированного массива вызывается без аргументов, он действует так, как если бы получил число 0. В результате создается типизированный массив, не способный хранить данные, потому что для буфера выделено ноль байт памяти.

Третий способ создания типизированных массивов — передача конструктору объекта в виде единственного аргумента. Таким объектом может быть:

- **Типизированный массив.** Все элементы этого массива будут скопированы в новые элементы нового типизированного массива. Например, если передать массив `int8` в конструктор `Int16Array`, значения `int8` будут скопированы в значения `int16` в новом массиве. Для нового типизированного массива будет выделен новый буфер.
- **Итерируемый объект.** Конструктор будет вызывать итератор объекта, чтобы получить элементы для вставки в типизированный массив. Если какой-то из элементов будет иметь значение, не совместимое с типом представления, конструктор возбудит ошибку.
- **Массив.** Элементы массива будут скопированы в новый типизированный массив. Если какой-то из элементов будет иметь значение, не совместимое с типом представления, конструктор возбудит ошибку.
- **Объект, подобный массиву.** Будет использован так же, как массив.

В каждом из этих случаев будет создан новый типизированный массив с данными из исходного объекта. Это может пригодиться, например, когда потребуется инициализировать типизированный массив некоторыми значениями, например:

```
let ints1 = new Int16Array([25, 50]),
    ints2 = new Int32Array(ints1);

console.log(ints1.buffer === ints2.buffer);    // false

console.log(ints1.byteLength);                  // 4
console.log(ints1.length);                      // 2
console.log(ints1[0]);                          // 25
console.log(ints1[1]);                          // 50

console.log(ints2.byteLength);                  // 8
console.log(ints2.length);                      // 2
console.log(ints2[0]);                          // 25
console.log(ints2[1]);                          // 50
```

Этот код создает `Int16Array` и инициализирует его массивом с двумя значениями. Затем создается `Int32Array` и инициализируется значениями из `Int16Array`. Значения 25 и 50 копируются из `ints1` в `ints2`, потому что эти два типизированных массива хранятся в разных буферах. Оба массива хранят одинаковые числа, но `ints2` занимает восемь байт, а `ints1` — только четыре.

РАЗМЕР ЭЛЕМЕНТА

Каждый типизированный массив состоит из множества элементов, и размер элемента – это количество байтов, составляющих каждый элемент. Данное значение хранится в свойстве `BYTES_PER_ELEMENT` каждого конструктора и каждого экземпляра, благодаря чему вы легко сможете узнать размер элемента:

```
console.log(UInt8Array.BYTES_PER_ELEMENT); // 1
console.log(UInt16Array.BYTES_PER_ELEMENT); // 2

let ints = new Int8Array(5);
console.log(ints.BYTES_PER_ELEMENT); // 1
```

Как показано в этом примере, вы можете прочитать значение свойства `BYTES_PER_ELEMENT` разных классов типизированных массивов, а также экземпляров этих классов.

Сходства типизированных и обычных массивов

Типизированные и обычные массивы имеют некоторые сходства, и вы уже видели в этой главе, что во многих ситуациях типизированные массивы используются подобно обычным массивам. Например, узнать количество элементов в типизированном массиве можно с помощью его свойства `length`, а элементы типизированного массива доступны непосредственно по числовым индексам:

```
let ints = new Int16Array([25, 50]);

console.log(ints.length); // 2
console.log(ints[0]); // 25
console.log(ints[1]); // 50

ints[0] = 1;
ints[1] = 2;

console.log(ints[0]); // 1
console.log(ints[1]); // 2
```

В этом примере создается новый массив `Int16Array` с двумя элементами. Чтение и запись элементов производятся с применением их числовых индексов, а присваиваемые им значения автоматически преобразуются в значения `int16`. Но на это сходства не заканчиваются.

ПРИМЕЧАНИЕ

В отличие от обычных массивов, размеры типизированных массивов нельзя изменить с помощью свойства `length`. Это свойство недоступно для записи, поэтому любая попытка изменить его просто игнорируется в нестрогом режиме и вызывает ошибку в строгом.

Общие методы

Типизированные массивы также включают большое количество методов, функционально эквивалентных одноименным методам обычных массивов. В типизированных массивах доступны следующие методы:

<code>copyWithin()</code>	<code>copyWithin()</code>	<code>copyWithin()</code>	<code>copyWithin()</code>
<code>findIndex()</code>	<code>findIndex()</code>	<code>findIndex()</code>	<code>findIndex()</code>
<code>lastIndexOf()</code>	<code>lastIndexOf()</code>	<code>lastIndexOf()</code>	<code>lastIndexOf()</code>
<code>slice()</code>	<code>slice()</code>	<code>slice()</code>	<code>slice()</code>
<code>entries()</code>	<code>entries()</code>	<code>entries()</code>	<code>entries()</code>

Имейте в виду, что, несмотря на функциональное сходство с аналогами в `Array.prototype`, эти методы действуют немного иначе. Методы типизированных массивов выполняют дополнительные проверки числовых типов, и возвращаемый массив является типизированным, а не обычным массивом (из-за действия свойства `Symbol.species`). Следующий простой пример демонстрирует разницу:

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => v * 2);

console.log(mapped.length);           // 2
console.log(mapped[0]);               // 50
console.log(mapped[1]);               // 100

console.log(mapped instanceof Int16Array); // true
```

Здесь для создания нового массива на основе значений в `ints` используется метод `map()`. Функция отображения удваивает каждое значение в массиве и возвращает новый массив `Int16Array`.

Те же самые итераторы

Типизированные массивы имеют те же три итератора, что и обычные массивы: метод `entries()`, метод `keys()` и метод `values()`. Это означает, что типизированные массивы, как и обычные массивы, можно использовать в циклах `for-of` и операторах расширения. Например:

```
let ints = new Int16Array([25, 50]),
    intsArray = [...ints];

console.log(intsArray instanceof Array); // true
console.log(intsArray[0]);               // 25
console.log(intsArray[1]);               // 50
```

Этот фрагмент создает новый массив с именем `intsArray`, содержащий те же данные, что и типизированный массив `ints`. Оператор расширения с легкостью преобразует типизированные массивы в обычные, как и любые другие итерируемые объекты.

Методы `of()` и `from()`

Кроме того, все типизированные массивы имеют статические методы `of()` и `from()`, действующие подобно методам `Array.of()` и `Array.from()`. Разница лишь в том, что методы типизированных массивов возвращают не обычные, а типизированные массивы. Ниже приводится несколько примеров использования этих методов для создания массивов:

```
let ints = Int16Array.of(25, 50),
    floats = Float32Array.from([1.5, 2.5]);
console.log(ints instanceof Int16Array); // true
console.log(floats instanceof Float32Array); // true
```

```
console.log(ints.length);           // 2
console.log(ints[0]);               // 25
console.log(ints[1]);               // 50

console.log(floats.length);        // 2
console.log(floats[0]);             // 1.5
console.log(floats[1]);             // 2.5
```

Методы `of()` и `from()` в этом примере создают массивы `Int16Array` и `Float32Array` соответственно. Эти методы позволяют создавать типизированные массивы так же просто, как обычные.

Различия типизированных и обычных массивов

Самое важное отличие типизированных массивов от обычных состоит в том, что типизированные массивы не являются обычными массивами. Типизированные массивы не наследуют тип `Array`, и `Array.isArray()` возвращает `false` для типизированных массивов. Например:

```
let ints = new Int16Array([25, 50]);

console.log(ints instanceof Array); // false
console.log(Array.isArray(ints));   // false
```

Переменная `ints` — типизированный массив, поэтому она не является экземпляром `Array` и не идентифицируется как массив. Это важное отличие, потому что, несмотря на сходства типизированных и обычных массивов, во многих случаях типизированные массивы действуют иначе.

Различия в поведении

Обычные массивы могут увеличиваться и сжиматься по мере операций с ними, но типизированные массивы всегда сохраняют свой размер неизменным. Нельзя присвоить значение элементу типизированного массива с несуществующим числовым индексом, как это возможно с обычными массивами, потому что типизированные массивы проигнорируют такую операцию. Например:

```
let ints = new Int16Array([25, 50]);

console.log(ints.length); // 2
console.log(ints[0]);     // 25
console.log(ints[1]);     // 50

ints[2] = 5;

console.log(ints.length); // 2
console.log(ints[2]);     // undefined
```

Несмотря на попытку присвоить число 5 элементу с индексом 2, размер массива `ints` не изменился. Значение свойства `length` осталось прежним, а присваиваемое значение было отброшено.

Типизированные массивы также выполняют проверки, чтобы гарантировать использование данных только допустимого типа. Любое недопустимое значение замещается нулем. Например:

```
let ints = new Int16Array(["hi"]);

console.log(ints.length); // 1
console.log(ints[0]);     // 0
```

Этот код пытается использовать строку "hi" для инициализации `Int16Array`. Строки являются недопустимым типом данных для типизированных массивов, поэтому вместо строки в массив добавляется значение 0. Свойство `length` массива получает значение 1, и даже при том, что элемент `ints[0]` существует, он получает значение 0.

Все методы, изменяющие значения в типизированных массивах, накладывают то же самое ограничение. Например, если в `map()` передать функцию, возвращающую значение, недопустимое для типизированного массива, вместо этого значения будет записан 0:

```
let ints = new Int16Array([25, 50]),
    mapped = ints.map(v => "hi");

console.log(mapped.length);           // 2
console.log(mapped[0]);               // 0
console.log(mapped[1]);               // 0

console.log(mapped instanceof Int16Array); // true
console.log(mapped instanceof Array);      // false
```

Поскольку строка "hi" не является 16-разрядным целым числом, вместо нее в массив результата будет записан 0. Благодаря такой коррекции ошибок методы типизированных массивов не возбуждают ошибок в присутствии недопустимых данных, потому что недопустимые данные никогда не попадают в массивы.

Отсутствующие методы

Несмотря на наличие у типизированных массивов большого количества тех же методов, что и у обычных массивов, у них отсутствуют некоторые типичные методы. Например, следующие методы не поддерживаются типизированными массивами:

<code>concat()</code>	<code>shift()</code>
<code>pop()</code>	<code>splice()</code>
<code>push()</code>	<code>unshift()</code>

Кроме метода `concat()`, все остальные методы могут изменять размер массива. Типизированные массивы имеют фиксированный размер, именно поэтому данные методы недоступны для типизированных массивов. Метод `concat()` не поддерживается, потому что результат конкатенации двух типизированных массивов (особенно если они хранят данные разных типов) выглядит сомнительным, и такая операция, прежде всего, противоречила бы целям применения типизированных массивов.

Дополнительные методы

Наконец, типизированные массивы имеют два метода, отсутствующие в обычных массивах: `set()` и `subarray()`. Эти два метода выполняют противоположные операции: `set()` копирует содержимое другого массива в существующий типизированный массив, а метод `subarray()` извлекает часть существующего типизированного массива в новый типизированный массив.

Метод `set()` принимает массив (типизированный или обычный) и необязательное смещение, определяющее, откуда начинать вставку данных. Если опустить второй аргумент, смещение получит значение по умолчанию, равное 0. Данные из массива-аргумента будут скопированы в данный типизированный массив, при этом гарантируется, что будут записаны данные только допустимых типов. Например:


```
let ints = new Int16Array(4);

ints.set([25, 50]);
ints.set([75, 100], 2);

console.log(ints.toString());    // 25,50,75,100
```

Этот код создает `Int16Array` с четырьмя элементами. Первый вызов `set()` скопирует два значения в первый и второй элементы массива. Второй вызов `set()` получает смещение 2, указывающее, что значения должны записываться в массив, начиная с третьего элемента.

Метод `subarray()` принимает необязательные начальный и конечный индексы (значение из элемента с конечным индексом не копируется, так же как в методе `slice()`) и возвращает новый типизированный массив. Можно опустить оба аргумента, чтобы создать полную копию исходного типизированного массива. Например:

```
let ints = new Int16Array([25, 50, 75, 100]),
    subints1 = ints.subarray(),
    subints2 = ints.subarray(2),
    subints3 = ints.subarray(1, 3);

console.log(subints1.toString());    // 25,50,75,100
console.log(subints2.toString());    // 75,100
console.log(subints3.toString());    // 50,75
```

В этом примере из оригинального массива `ints` создаются три типизированных массива. Массив `subints1` — полная копия массива `ints`, то есть содержит то же количество элементов с теми же значениями. Так как данные в массив `subints2` копируются, начиная с индекса 2, в нем оказываются только два последних элемента из `ints` (75 и 100). Массив `subints3` содержит только два средних элемента из массива `ints`, потому что метод `subarray()` был вызван с двумя аргументами — начальным и конечным индексами.

В заключение

Спецификация ECMAScript 6 продолжила курс, начатый в ECMAScript 5, на увеличение практической ценности массивов. Появились два новых способа создания массивов: методы `Array.of()` и `Array.from()`. Метод `Array.from()` может преобразовывать в массивы итерируемые объекты и объекты, подобные массивам. Оба метода наследуются производными классами и используют свойство `Symbol.species` для определения типа возвращаемого значения (другие наследуемые методы, возвращающие массивы, также используют свойство `Symbol.species`).

У массивов появилось и несколько новых методов. Методы `fill()` и `copyWithin()` дают возможность заменять элементы массива. Методы `find()` и `findIndex()` удобно использовать для поиска первого элемента в массиве, соответствующего некоторому критерию. Метод `find()` возвращает первый элемент, соответствующий критерию, а `findIndex()` возвращает индекс.

Типизированные массивы формально не являются массивами, потому что не наследуют тип `Array`, но они выглядят и действуют как массивы. Типизированные массивы могут содержать данные любого из восьми разных числовых типов и размещаются в объектах `ArrayBuffer`, представляющих биты числа или последовательности чисел. Типизированные массивы обеспечивают более эффективный способ выполнения арифметических операций, потому что значения не преобразуются взад и вперед между форматами, как это происходит при использовании числового типа JavaScript.

Объект Promise и асинхронное программирование

Одна из самых мощных особенностей JavaScript — простота асинхронного программирования. Как язык, созданный для Всемирной паутины, JavaScript должен был иметь возможность отвечать на асинхронные действия пользователя, такие как щелчки мышью и нажатия клавиш. Фреймворк Node.js способствует применению приемов асинхронного программирования на JavaScript с применением обратных вызовов как альтернативы событиям. Но с увеличением количества программ, использующих приемы асинхронного программирования, стало очевидно, что события и обратные вызовы не обладают достаточной широтой возможностей для удовлетворения всех потребностей. Для решения этой проблемы были разработаны объекты асинхронных вычислений **Promise**.

Объекты **Promise** — еще одна разновидность асинхронного программирования, и они действуют подобно своим аналогам в других языках. Так же как события и обратные вызовы, объект **Promise** определяет некоторый код, который должен быть выполнен позднее, но они также явно сообщают, был ли этот код выполнен успешно или потерпел неудачу. Объекты **Promise** можно объединять в цепочки в зависимости от успеха или неудачи, что сделает ваш код более легким для чтения и отладки.

В этой главе рассказывается о том, как работают объекты **Promise**. Однако для полного понимания важно знать некоторые основные понятия, на которых основываются объекты асинхронных вычислений.

Основы асинхронного программирования

Движки JavaScript опираются на понятие однопоточного цикла событий. Под *однопоточным* (*single-threaded*) здесь подразумевается, что в каждый конкретный момент времени выполняется только один фрагмент кода. Сравните это с другими языками, такими как Java или C++, где одновременно может действовать несколько потоков, выполняющих разные фрагменты кода. Поддержка и защита состояния, когда сразу несколько фрагментов кода могут читать и изменять это состояние, является сложной задачей и нередко — источником ошибок в многопоточном программном обеспечении.

Движки JavaScript могут выполнять одновременно только один фрагмент кода, поэтому они должны следить за тем, какой код предназначен для выполнения. Этот код сохраняется в очереди заданий. Всякий раз, когда появляется фрагмент кода, готовый к выполнению, он добавляется в *очередь заданий*. Когда движок JavaScript завершает выполнение кода, цикл событий выполняет следующее задание из очереди. *Цикл событий* — это процесс внутри движка JavaScript, который следит за выполнением кода и управляет очередью заданий. Имейте в виду, что поскольку задания находятся в очереди, они выполняются последовательно, от первого к последнему.

Модель событий

Когда пользователь щелкает на кнопке или нажимает клавишу на клавиатуре, генерируется событие, такое как `onclick`. В ответ на действие пользователя это событие может добавить новое задание в конец очереди. Это самая простая форма асинхронного программирования на JavaScript. Код обработчика событий не будет выполнен, пока не возникнет событие, а когда он выполняется, получает соответствующий контекст. Например:

```
let button = document.getElementById("my-btn");
button.onclick = function(event) {
    console.log("Clicked");
};
```

В этом примере вызов `console.log("Clicked")` не произойдет, пока пользователь не щелкнет на кнопке. Когда произойдет щелчок, функция, присвоенная свойству `onclick`, добавляется в очередь заданий и будет выполнена по завершении всех предшествующих ей заданий.

События прекрасно подходят для простых взаимодействий, но объединение в цепочку нескольких разных асинхронных вызовов весьма усложняет задачу из-за необходимости правильно определить цель (`button` в этом примере) для каждого события. Кроме того, вы должны гарантировать, что все обработчики событий будут добавлены к моменту, когда событие возникнет в первый раз. Например, если щелчок на кнопке произойдет до того, как будет установлен обработчик события `onclick`, ничего не произойдет. События удобно использовать для организации отклика в ответ на действия пользователя и другие нечастые происшествия, но они оказываются слишком негибкими для более сложных ситуаций.

Обратные вызовы

Фреймворк Node.js продвигает улучшенную модель асинхронного программирования, основанную на обратных вызовах. Шаблон обратных вызовов напоминает модель событий, потому что в нем асинхронный код не выполняется до некоторой точки времени в будущем. И отличается тем, что функция для вызова передается в аргументе, как показано ниже:

```
readFile("example.txt", function(err, contents) {
    if (err) {
        throw err;
    }

    console.log(contents);
});

console.log("Hi!");
```

В этом примере демонстрируется традиционный для Node.js стиль оформления функций обратного вызова — *сначала ошибка (error-first)*. Функция `readFile()` должна прочитать файл с диска (имя файла передается в первом аргументе) и затем выполнить обратный вызов (вызвать функцию во втором аргументе). Если в процессе чтения файла возникла ошибка, объект ошибки передается функции обратного вызова в аргументе `err`; в противном случае в аргументе `contents` передается содержимое файла.

Функция `readFile()`, использующая шаблон обратных вызовов, немедленно начинает выполняться и приостанавливается, когда начинается чтение файла с диска. Это означает, что `console.log("Hi!")` выведет свою строку сразу после вызова `readFile()`, но до того, как `console.log(contents)` выведет хоть что-то. Когда `readFile()` завершится, она добавит в конец очереди новое задание с функцией обратного вызова и ее аргументами. Это задание выполнится сразу, как только будут выполнены все предшествующие задания.

Шаблон обратных вызовов более гибкий, чем события, потому что составление цепочек из обратных вызовов реализуется проще. Например:

```
readFile("example.txt", function(err, contents) {
  if (err) {
    throw err;
  }

  writeFile("example.txt", function(err) {
    if (err) {
      throw err;
    }

    console.log("File was written!");
  });
});
```

В этом примере успешный вызов `readFile()` приводит к другому асинхронному вызову — на этот раз функции `writeFile()`. Обратите внимание, что в обеих функциях используется один и тот же шаблон проверки `err`. Когда `readFile()` завершится, она добавит в очередь задание, которое вызовет функцию `writeFile()`, если не будет обнаружено ошибок. Затем `writeFile()` добавит задание в очередь, когда завершит работу.

Этот шаблон работает довольно хорошо, но вы быстро заметите, что благодаря шаблону попали в *ад обратных вызовов* (*callback hell*). Ад обратных вызовов возникает, когда приходится писать слишком много обратных вызовов, вложенных друг в друга, например:

```
method1(function(err, result) {

  if (err) {
    throw err;
  }

  method2(function(err, result) {

    if (err) {
      throw err;
    }

    method3(function(err, result) {

      if (err) {
        throw err;
      }

      method4(function(err, result) {

        if (err) {
          throw err;
        }

        method5(result);
      });
    });
  });
});
```

```
});  
});
```

В результате вложения нескольких вызовов методов, как в данном примере, код получается запутанным, трудным для изучения и отладки. Кроме того, обратные вызовы порождают проблемы, когда требуется реализовать более сложную функциональность. Например, как быть, если желательно, чтобы две асинхронные операции выполнялись параллельно и уведомляли основную программу о своем завершении? Как быть, если требуется одновременно запустить две асинхронные операции, но использовать результат только той, что завершилась первой? В таких случаях придется отслеживать выполнение большого количества обратных вызовов и операций уборки мусора, и такие ситуации существенно упрощают объекты `Promise`.

Основы объектов Promise

Объект `Promise` служит хранилищем для результата асинхронной операции. Вместо подписки на событие или передачи обратного вызова в функцию можно просто вернуть объект `Promise`, как показано ниже:

```
// readFile "обещает" (promises) завершить выполнение  
// в некоторый момент в будущем  
let promise = readFile("example.txt");
```

Здесь функция `readFile()` не начнет читать файл немедленно — это произойдет позднее. Вместо этого функция возвращает объект `Promise`, представляющий асинхронную операцию чтения, результат которой вы сможете обработать в будущем. Фактическая обработка результата будет зависеть от того, как завершится жизненный цикл объекта `Promise`.

Жизненный цикл объекта Promise

Каждый объект `Promise` проходит короткий жизненный цикл, начиная с *состояния ожидания*, которое указывает, что асинхронная операция еще не завершилась. Объект `Promise` в состоянии ожидания считается *неустановившимся*. В предыдущем примере функция `readFile()` возвращает объект `Promise` в состоянии ожидания. Как только асинхронная операция завершится, объект `Promise` считается *установившимся* и переходит в одно из двух возможных состояний:

- **Выполнено.** Асинхронная операция выполнена успешно.
- **Отклонено.** Асинхронная операция завершилась неудачей из-за ошибки или по какой-то другой причине.

Внутреннее свойство `[[PromiseState]]` получает значение `"pending"` (ожидание), `"fulfilled"` (выполнено) или `"rejected"` (отклонено), отражающее состояние объекта. Это свойство не экспортируется объектами `Promise`, поэтому определить их состояние программно невозможно. Но с помощью метода `then()` можно запрограммировать выполнение некоторых действий, которые должны выполняться при смене состояния объектом `Promise`.

Метод `then()` имеется во всех объектах `Promise` и принимает два аргумента. Первый аргумент — функция, которая должна быть вызвана после успешного выполнения асинхронной операции. Ей будут переданы все данные, связанные с асинхронной операцией. Второй аргумент — функция, которая должна быть вызвана в случае неудачи. Ей также будут переданы все данные, описывающие причину неудачи.

ПРИМЕЧАНИЕ

Любой объект, реализующий метод `then()`, описанный в предыдущем абзаце, называется *thenable-объектом*¹. Все объекты `Promise` являются *thenable-объектами*, но не все *thenable-объекты* представляют асинхронные операции.

¹ Дословно: «объект с методом `then()`». Этот термин используется в спецификации ECMAScript 6. – *Примеч. пер.*

Оба аргумента метода `then()` являются необязательными, то есть вы можете передавать ему любые комбинации обработчиков успешного и неуспешного завершения операции. Например, взгляните на следующие вызовы метода `then()`:

```
let promise = readFile("example.txt");
```

```
promise.then(function(contents) {  
    // выполнено  
    console.log(contents);  
}, function(err) {  
    // отклонено  
    console.error(err.message);  
});
```

```
promise.then(function(contents) {  
    // выполнено  
    console.log(contents);  
});
```

```
promise.then(null, function(err) {  
    // отклонено  
    console.error(err.message);  
});
```

Все три вызова `then()` оперируют с одним и тем же объектом `promise`. Первый готов обработать успешное и неуспешное завершение операции. Второй обрабатывает только успешное завершение; ошибки будут просто игнорироваться. Третий обрабатывает только неуспешное завершение; успешное завершение игнорируется.

Объекты `Promise` имеют также метод `catch()`, действующий подобно методу `then()`, когда определяется только обработчик отказа. Например, следующие вызовы `catch()` и `then()` функционально эквивалентны:

```
promise.catch(function(err) {  
    // отклонено  
    console.error(err.message);  
});  
// то же самое:  
promise.then(null, function(err) {  
    // отклонено  
    console.error(err.message);  
});
```

Методы `then()` и `catch()` предназначены для совместного использования с целью правильной обработки результата асинхронной операции. Эта система лучше, чем использование событий и обратных вызовов, потому что ясно указывает, как завершилась операция: успешно или неуспешно. (Обычно события не возбуждаются в случае ошибки, а в обратных вызовах надо помнить о

необходимости проверять аргумент, представляющий ошибку.) Но помните, что если не указать обработчик отказа, все ошибки будут просто игнорироваться. Всегда подключайте обработчик отказа, даже если он просто выводит сообщения об ошибках в консоль.

Обработчик успешного или неуспешного завершения операции будет вызван, даже если подключить его после того, как объект `Promise` установится. Это позволяет добавлять обработчики в любой момент с гарантией, что они будут вызваны. Например:

```
let promise = readFile("example.txt");

// первоначальный обработчик успешного завершения
promise.then(function(contents) {
  console.log(contents);

  // добавить еще один обработчик
  promise.then(function(contents) {
    console.log(contents);
  });
});
```

В этом примере обработчик успешного завершения добавляет еще один обработчик успешного завершения в тот же объект `Promise`. В этот момент асинхронная операция уже успешно завершилась, поэтому новый обработчик добавляется в очередь заданий и будет вызван после выполнения всех предшествующих заданий. Обработчики отказа действуют точно так же.

ПРИМЕЧАНИЕ

Каждый вызов `then()` или `catch()` создает новое задание для выполнения после того, как объект `Promise` будет установлен. Но эти задания помещаются в отдельную очередь, предназначенную специально для нужд объектов `Promise`. Знание всех деталей работы этой второй очереди заданий не требуется для понимания особенностей использования объектов `Promise`. Достаточно понимать, как вообще действует очередь заданий.

Создание неустановившихся объектов `Promise`

Новые объекты `Promise` создаются с помощью конструктора `Promise`. Он принимает единственный аргумент: функцию, которую называют *исполнителем*, содержащую код инициализации объекта. Исполнителю передаются две функции с именами `resolve()` и `reject()`. Функция `resolve()` вызывается, когда исполнитель завершается успехом, чтобы сообщить, что объект готов к установке, а функция `reject()` сообщает, что исполнитель потерпел неудачу.

Ниже приводится пример использования `Promise` в Node.js для реализации функции `readFile()`, которую вы видели выше в этой главе:

```
// Пример из Node.js

let fs = require("fs");

function readFile(filename) {
  return new Promise(function(resolve, reject) {

    // запустить асинхронную операцию
    fs.readFile(filename, { encoding: "utf8" }, function(err, contents) {
```

```
        // проверить наличие ошибки
        if (err) {
            reject(err);
            return;
        }

        // если файл прочитан успешно
        resolve(contents);
    });
});
}
```

```
let promise = readFile("example.txt");
```

```
// подключить оба обработчика
promise.then(function(contents) {
    // выполнено
    console.log(contents);
}, function(err) {
    // отклонено
    console.error(err.message);
});
```

В этом примере встроенный в Node.js асинхронный вызов `fs.readFile()` заключен в объект `Promise`. Исполнитель передает либо объект ошибки в вызов `reject()`, либо содержимое файла в вызов `resolve()`.

Имейте в виду, что исполнитель запускается сразу же, как только вызывается `readFile()`. Когда исполнитель вызывает `resolve()` или `reject()`, в очередь добавляется задание, которое выполнит установку объекта `Promise`. Это называется *планированием задания*, и если вам пришлось использовать функцию `setTimeout()` или `setInterval()`, вы должны уже быть знакомы с этим понятием. Планируя задание, вы добавляете новое задание в очередь, как бы говоря: «Выполнить его не прямо сейчас, а позднее». Например, функция `setTimeout()` позволяет указать задержку перед добавлением задания в очередь:

```
// добавить эту функцию в очередь заданий через 500 мс
setTimeout(function() {
    console.log("Timeout");
}, 500)
```

```
console.log("Hi!");
```

Этот код запланирует добавление задания в очередь через 500 мс. Два вызова `console.log()` выведут следующее:

```
Hi!
Timeout
```

Благодаря задержке в 500 мс вывод функции, переданной в `setTimeout()`, появился после текста, который вывел вызов `console.log("Hi!")`.

Объекты `Promise` действуют похожим образом. Исполнитель объекта `Promise` выполняется немедленно, то есть прежде, чем выполнится любой код, находящийся ниже в исходном тексте сценария. Например:


```
let promise = new Promise(function(resolve, reject) {  
    console.log("Promise");  
    resolve();  
});
```

```
console.log("Hi!");
```

Этот фрагмент выведет:

```
Promise  
Hi!
```

Вызов `resolve()` запустит асинхронную операцию. Функции, переданные в `then()` и `catch()`, выполняются асинхронно, потому что также добавляются в очередь заданий. Например:

```
let promise = new Promise(function(resolve, reject) {  
    console.log("Promise");  
    resolve();  
});
```

```
promise.then(function() {  
    console.log("Resolved.");  
});
```

```
console.log("Hi!");
```

Этот пример выведет:

```
Promise  
Hi!  
Resolved
```

Обратите внимание: хотя вызов `then()` находится выше строки с инструкцией `console.log("Hi!")`, это не значит, что он будет выполнен раньше (в отличие от исполнителя). Причина в том, что обработчики успешного и неуспешного выполнения операции всегда добавляются в конец очереди заданий — после того, как исполнитель завершится.

Создание установившихся объектов Promise

Конструктор `Promise` отлично подходит для создания неустановившихся объектов `Promise` благодаря динамической природе исполнителей. Но если необходимо создать объект `Promise`, представляющий единственное известное значение, нет смысла планировать задание, которое просто передаст значение в вызов функции `resolve()`. Вместо этого можно воспользоваться одним из двух методов, чтобы создать установившийся объект `Promise` с заданным значением.

Использование `Promise.resolve()`

Метод `Promise.resolve()` принимает единственный аргумент и возвращает объект `Promise` в состоянии «выполнено». Значит, в данном случае не происходит планирования задания, и вам нужно добавить один или несколько обработчиков успешного выполнения, чтобы извлечь значение из объекта `Promise`. Например:

```
let promise = Promise.resolve(42);

promise.then(function(value) {
  console.log(value);    // 42
});
```

Этот фрагмент кода создает установившийся объект **Promise**, поэтому обработчик успешного выполнения получит значение 42. Если в этом примере определить обработчик неудачи, он никогда не будет вызван, потому что данный объект **Promise** никогда не окажется в состоянии «отклонено».

Использование **Promise.reject()**

Аналогично можно создавать объекты **Promise** в состоянии «отклонено», используя метод **Promise.reject()**. Он действует подобно методу **Promise.resolve()**, за исключением того, что создает объект **Promise** в состоянии «отклонено», как показано ниже:

```
let promise = Promise.reject(42);

promise.catch(function(value) {
  console.log(value);    // 42
});
```

Любые дополнительные обработчики неудачи, добавленные в этот объект **Promise**, также будут вызваны, но не обработчики успешного выполнения.

ПРИМЕЧАНИЕ

Если в вызов **Promise.resolve()** или **Promise.reject()** передать объект **Promise**, они вернут его без изменений.

Thenable-объекты, отличные от **Promise**

Оба метода, **Promise.resolve()** и **Promise.reject()**, принимают также любые thenable-объекты, не являющиеся объектами **Promise**. Получая такой thenable-объект, эти методы создают новый объект **Promise**, который вызывается после функции **then()**.

Thenable-объект, не являющийся объектом **Promise**, создается, когда объект имеет метод **then()** с аргументами **resolve** и **reject**, например:

```
let thenable = {
  then: function(resolve, reject) {
    resolve(42);
  }
};
```

В этом примере thenable-объект не обладает никакими характеристиками, присущими объектам **Promise**, кроме метода **then()**. Вызовом **Promise.resolve()** такой объект можно превратить в установившийся объект **Promise** в состоянии «выполнено»:

```
let thenable = {
  then: function(resolve, reject) {
    resolve(42);
  }
};
```

```
let p1 = Promise.resolve(thenable);
p1.then(function(value) {
  console.log(value);    // 42
});
```

В этом примере `Promise.resolve()` вызовет `thenable.then()`, чтобы определить состояние объекта. В данном случае `thenable` получит состояние «выполнено», потому что внутри метода `then()` вызывается `resolve(42)`. Новый объект `Promise` с именем `p1` создается сразу в состоянии «выполнено» со значением, полученным из `thenable` (то есть 42), и обработчик успешного выполнения в `p1` получит значение 42.

Ту же процедуру с вызовом `Promise.resolve()` можно использовать, чтобы создать объект `Promise` в состоянии «отклонено»:

```
let thenable = {
  then: function(resolve, reject) {
    reject(42);
  }
};

let p1 = Promise.resolve(thenable);
p1.catch(function(value) {
  console.log(value);    // 42
});
```

Этот пример похож на предыдущий, за исключением того, что `thenable` получает состояние «отклонено». Когда выполняется `thenable.then()`, создается новый объект `Promise` в состоянии «отклонено» и со значением 42. Это значение затем передается обработчику отказа в объекте `p1`.

Методы `Promise.resolve()` и `Promise.reject()` реализованы так, чтобы дать возможность использовать `thenable`-объекты, не являющиеся объектами `Promise`. Многие библиотеки использовали `thenable`-объекты еще до введения объектов `Promise` в ECMAScript 6, поэтому возможность преобразования `thenable`-объектов в официально поддерживаемые объекты `Promise` очень важна для сохранения обратной совместимости с ранее существовавшими библиотеками. Если вы сомневаетесь, что объект является объектом `Promise`, передавайте его через `Promise.resolve()` или `Promise.reject()` (в зависимости от ожидаемого результата), потому что настоящие объекты `Promise` будут переданы без изменений.

Ошибки исполнителя

Если ошибка возникнет внутри исполнителя, будет вызван обработчик отказа в объекте `Promise`. Например:

```
let promise = new Promise(function(resolve, reject) {
  throw new Error("Explosion!");
});

promise.catch(function(error) {
  console.log(error.message);    // "Explosion!"
});
```

В этом примере исполнитель преднамеренно возбуждает ошибку. Каждый исполнитель неявно вызывает операцию в блоке `try-catch`, поэтому данная ошибка будет перехвачена и передана в обработчик отказа. Ниже приводится эквивалент предыдущего примера:

```
let promise = new Promise(function(resolve, reject) {
  try {
    throw new Error("Explosion!");
  } catch (ex) {
    reject(ex);
  }
});

promise.catch(function(error) {
  console.log(error.message); // "Explosion!"
});
```

Исполнитель перехватывает любые ошибки, но сообщает о них, только если назначен обработчик отказа. В противном случае ошибки просто игнорируются. На ранних этапах, когда поддержка объектов `Promise` только появилась, это вызывало проблемы у разработчиков, поэтому в JavaScript были добавлены точки подключения обработчиков для перехвата отклоненных объектов `Promise`.

Глобальная обработка отклоненных объектов Promise

Один из самых противоречивых аспектов объектов `Promise` — игнорирование ошибок, возникающих, когда `Promise` терпит неудачу в отсутствие обработчика отказа. Некоторые считают это самым большим недостатком спецификации, потому что это единственный раздел языка JavaScript, который прячет ошибки.

Из-за природы объектов `Promise` очень непросто определить, была ли обработана ошибка, возникшая в процессе их выполнения. Например, взгляните на следующий пример:

```
let rejected = Promise.reject(42);

// в этой точке ошибка не обработана

// немного позднее...
rejected.catch(function(value) {
  // теперь ошибка обработана
  console.log(value);
});
```

В любом месте можно вызвать `then()` или `catch()` и правильно обработать оба состояния объекта `Promise`, но с таким подходом трудно понять, когда точно будет обработан объект. В данном случае асинхронная операция отклоняется немедленно, но обрабатывается она не сразу.

Вполне возможно, что в будущей версии ECMAScript эта проблема будет исправлена, а пока Node.js и браузеры предлагают свои решения. Они не являются частью спецификации ECMAScript 6, но оказывают ценную помощь при работе с объектами `Promise`.

Обработка отказов в Node.js

Node.js генерирует два события в объекте `process`, связанные с обработкой отказа в `Promise`:

- `unhandledRejection`. Генерируется, когда `Promise` терпит неудачу и в ближайшей итерации цикла обработки событий не происходит вызова соответствующего обработчика.
- `rejectionHandled`. Генерируется, когда `Promise` терпит неудачу и соответствующий обработчик вызывается спустя одну итерацию цикла обработки событий.

Эта пара событий — назначается с целью помочь выявить асинхронные операции, завершившиеся ошибкой и не обработанные.

Обработчику события `unhandledRejection` передается причина ошибки (обычно объект ошибки) и объект `Promise`, в котором возникла ошибка. Следующий пример демонстрирует обработку события `unhandledRejection`:

```
let rejected;
process.on("unhandledRejection", function(reason, promise) {
  console.log(reason.message);          // "Explosion!"
  console.log(rejected === promise);    // true
});
```

```
rejected = Promise.reject(new Error("Explosion!"));
```

Этот код создает объект `Promise` в состоянии «отклонено» с объектом ошибки и ожидает события `unhandledRejection`. Обработчик события получает объект ошибки в первом аргументе и объект `Promise` — во втором.

Обработчик события `rejectionHandled` имеет только один аргумент — объект `Promise`, в котором возникла ошибка. Например:

```
let rejected;

process.on("rejectionHandled", function(promise) {
  console.log(rejected === promise);    // true
});

rejected = Promise.reject(new Error("Explosion!"));

// выдержать паузу перед добавлением обработчика ошибки
setTimeout(function() {
  rejected.catch(function(value) {
    console.log(value.message);          // "Explosion!"
  });
}, 1000);
```

Здесь событие `rejectionHandled` возбуждается, когда вызов обработчика отказа наконец-то произойдет. Если бы обработчик отказа был подключен к объекту `rejected` непосредственно после его создания, событие не было бы возбуждено. Вместо этого вызов обработчика отказа произошел бы в той же итерации цикла обработки событий, в которой был создан объект `rejected`, что не особенно полезно.

Чтобы отследить потенциально необработанные отказы, используйте события `rejectionHandled` и `unhandledRejection` для сохранения списка потенциально необработанных отказов. Затем выдержите некоторую паузу и проверьте список. Например, взгляните на следующий простой способ выявления необработанных отказов:

```
let possiblyUnhandledRejections = new Map();

// добавить в ассоциативный массив вновь появившийся необработанный отказ
process.on("unhandledRejection", function(reason, promise) {
  possiblyUnhandledRejections.set(promise, reason);
});

process.on("rejectionHandled", function(promise) {
  possiblyUnhandledRejections.delete(promise);
});
```

```
});

setInterval(function() {

    possiblyUnhandledRejections.forEach(function(reason, promise) {
        console.log(reason.message ? reason.message : reason);

        // обработать отказы
        handleRejection(promise, reason);
    });

    possiblyUnhandledRejections.clear();

}, 60000);
```

Этот код использует ассоциативный массив для хранения объектов `Promise` и соответствующих им ошибок, описывающих причину отказа. Объекты `Promise` служат ключами, а объекты ошибок — связанными с ними значениями. Каждый раз когда возникает событие `unhandledRejection`, объект `Promise` с причиной отказа добавляется в ассоциативный массив. Каждый раз когда возникает событие `rejectionHandled`, объект `Promise`, отказ которого был обработан, удаляется из ассоциативного массива. В результате `possiblyUnhandledRejections` увеличивается и сжимается по мере появления событий. Вызов `setInterval()` периодически проверяет список потенциально необработанных отказов и выводит информацию в консоль (в действующем приложении можно сделать что-то еще, чтобы записать информацию об отказе в журнал или как-то обработать его). В этом примере используется ассоциативный массив, а не ассоциативный массив со слабыми ссылками, потому что его нужно просматривать периодически, чтобы узнать, какие объекты `Promise` в нем присутствуют, что невозможно при использовании ассоциативного массива со слабыми ссылками.

Этот пример использует характерные особенности Node.js, тем не менее в браузерах имеется аналогичный механизм, позволяющий извещать разработчиков о необработанных отказах.

Обработка отказов в браузерах

Браузеры также возбуждают два события, помогающие идентифицировать необработанные отказы. Эти события возбуждаются объектом `window` и фактически эквивалентны своим аналогам в Node.js:

`unhandledrejection`. Генерируется, когда `Promise` терпит неудачу и в ближайшей итерации цикла обработки событий не происходит вызова соответствующего обработчика.

`rejectionhandled`. Генерируется, когда `Promise` терпит неудачу и соответствующий обработчик вызывается спустя одну итерацию цикла обработки событий.

В отличие от Node.js, где обработчику события передаются отдельные параметры, в браузерах обработчики этих событий получают объект события со следующими свойствами:

`type`. Имя события ("unhandledrejection" или "rejectionhandled").

`promise`. Объект `Promise`, потерпевший неудачу.

`reason`. Значение отказа из объекта `Promise`.

Еще одно отличие реализации событий в браузерах — значение отказа (`reason`) доступно в обоих событиях. Например:

```
let rejected;

window.onunhandledrejection = function(event) {
    console.log(event.type);           // "unhandledrejection"
    console.log(event.reason.message); // "Explosion!"
}
```

```
    console.log(rejected === event.promise);    // true
  });

window.onrejectionhandled = function(event) {
  console.log(event.type);                      // "rejectionhandled"
  console.log(event.reason.message);           // "Explosion!"
  console.log(rejected === event.promise);     // true
};

rejected = Promise.reject(new Error("Explosion!"));
```

В этом примере установка обработчиков событий `onunhandledrejection` и `onrejectionhandled` производится с использованием нотации DOM Level 0. (При желании можно также использовать `addEventListener("unhandledrejection")` и `addEventListener("rejectionhandled")`.) Каждый обработчик получает объект события с информацией об отказе в объекте `Promise`. Свойства `type`, `promise` и `reason` доступны обоим обработчикам.

Код, отслеживающий необработанные отказы в браузере, тоже очень похож на код для Node.js:

```
let possiblyUnhandledRejections = new Map();

// добавить в ассоциативный массив вновь появившийся необработанный отказ
window.onunhandledrejection = function(event) {
  possiblyUnhandledRejections.set(event.promise, event.reason);
};

window.onrejectionhandled = function(event) {
  possiblyUnhandledRejections.delete(event.promise);
};

setInterval(function() {

  possiblyUnhandledRejections.forEach(function(reason, promise) {
    console.log(reason.message ? reason.message : reason);

    // обработать отказы
    handleRejection(promise, reason);
  });

  possiblyUnhandledRejections.clear();

}, 60000);
```

Эта реализация практически повторяет реализацию для Node.js. В ней используется тот же подход к сохранению объектов `Promise` и соответствующих им значений отказов в ассоциативном массиве и последующему их исследованию. Единственное существенное отличие — порядок извлечения информации в обработчиках событий.

Обработка отказов в объектах `Promise` может оказаться очень непростым делом, но вы только начинаете познавать их настоящий потенциал. Теперь пришло время сделать следующий шаг и объединить в цепочку несколько объектов `Promise`.

Составление цепочек из объектов Promise

В данный момент объекты `Promise` могут выглядеть чуть больше, чем небольшая модернизация приема, основанного на применении комбинации обратного вызова и функции `setTimeout()`, но в действительности объекты `Promise` — это намного больше, чем кажется на первый взгляд. В частности, существует несколько способов объединения объектов `Promise` в цепочки для реализации более сложного асинхронного поведения.

Каждый вызов `then()` или `catch()` фактически создает и возвращает новый объект `Promise`. Этот второй объект `Promise` остается в состоянии ожидания, пока первый не перейдет в состояние «выполнено» или «отклонено». Взгляните на следующий пример:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

p1.then(function(value) {
  console.log(value);
}).then(function() {
  console.log("Finished");
});
```

Этот код выведет:

```
42
Finished
```

Вызов `p1.then()` вернет второй объект `Promise`, для которого также будет вызван метод `then()`. Обработчик успешного выполнения во втором вызове `then()` будет вызван, только если первый объект `Promise` перейдет в состояние «выполнено». Если рассоединить цепочку в этом примере, то же самое можно было бы реализовать так:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = p1.then(function(value) {
  console.log(value);
})

p2.then(function() {
  console.log("Finished");
});
```

В этой новой версии кода результат вызова `p1.then()` сохраняется в `p2`, а затем вызывается `p2.then()`, чтобы добавить заключительный обработчик успешного выполнения. Как вы уже могли догадаться, вызов `p2.then()` также возвращает объект `Promise`, но в данном примере он не используется.

Перехват ошибок

Составление цепочек из объектов `Promise` дает возможность перехватывать ошибки, возникающие в обработчиках успешного или неуспешного выполнения операции, подключенных к предыдущему объекту `Promise`. Например:


```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

p1.then(function(value) {
  throw new Error("Boom!");
}).catch(function(error) {
  console.log(error.message);    // "Boom!"
});
```

В этом примере обработчик успешного выполнения операции, подключенный к `p1`, возбуждает ошибку. Связанный вызов метода `catch()` второго объекта `Promise` позволяет перехватить эту ошибку в обработчике отказа. То же верно и в отношении обработчика отказа, возбуждающего ошибку:

```
let p1 = new Promise(function(resolve, reject) {
  throw new Error("Explosion!");
});

p1.catch(function(error) {
  console.log(error.message);    // "Explosion!"
  throw new Error("Boom!");
}).catch(function(error) {
  console.log(error.message);    // "Boom!"
});
```

Здесь исполнитель возбуждает ошибку, что приводит к вызову обработчика отказа в объекте `p1`. Этот обработчик возбуждает другую ошибку, которая перехватывается обработчиком отказа второго объекта `Promise`. Объекты `Promise`, объединенные в цепочку, распознают ошибки, возникшие в других объектах `Promise`, предшествующих им в цепочке.

ПРИМЕЧАНИЕ

Всегда добавляйте обработчик отказа в конец цепочки объектов `Promise`, чтобы гарантировать правильную обработку любых возникающих ошибок.

Возврат значений в цепочке объектов Promise

Еще одним важным аспектом цепочек объектов `Promise` является возможность передачи данных от одного объекта следующему. Выше уже было показано, как значение, переданное в обработчик `resolve()` внутри исполнителя, передается обработчику успешного выполнения операции этого объекта `Promise`, однако данные можно также передавать вдоль по цепочке, определяя возвращаемое значение в обработчике успешного выполнения. Например:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

p1.then(function(value) {
  console.log(value);    // "42"
  return value + 1;
}).then(function(value) {
  console.log(value);    // "43"
});
```

Обработчик успешного выполнения операции для объекта `p1` возвращает `value + 1`. Так как `value` получает значение 42 (от исполнителя), обработчик возвращает 43. Это значение затем передается обработчику успешного выполнения второго объекта `Promise`, который выводит его в консоль.

То же самое можно проделать с обработчиком отказа. Когда обработчик отказа вызывается, он также может вернуть значение. В этом случае возвращаемое значение будет передано обработчику успешного выполнения следующего в цепочке объекта `Promise`, как показано ниже.

```
let p1 = new Promise(function(resolve, reject) {
  reject(42);
});

p1.catch(function(value) {
  // обработчик отказа первого объекта
  console.log(value);    // "42"
  return value + 1;
}).then(function(value) {
  // обработчик успешного выполнения второго объекта
  console.log(value);    // "43"
});
```

Здесь исполнитель вызывает `reject()` с аргументом 42. Это значение передается обработчику отказа первого объекта `Promise`, который возвращает `value + 1`. Несмотря на то что возвращаемое значение исходит из обработчика отказа, для его получения все еще используется обработчик успешного выполнения следующего в цепочке объекта `Promise`. Таким способом можно обработать отказ в одном объекте `Promise` и восстановить нормальную работу всей цепочки.

Возврат объектов Promise в цепочке

Возврат простых значений из обработчиков успешного или неуспешного выполнения операции позволяет передавать данные между объектами `Promise`, но как поступить, если потребуется вернуть объект? Если возвращаемым объектом является объект `Promise`, необходимо сделать еще один шаг, чтобы определить, как его обрабатывать. Взгляните на следующий пример:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
  resolve(43);
});

p1.then(function(value) {
  // первый обработчик успешного выполнения
  console.log(value);    // 42
  return p2;
}).then(function(value) {
  // второй обработчик успешного выполнения
  console.log(value);    // 43
});
```

В этом примере `p1` планирует задание, которое успешно выполняется и возвращает 42. Обработчик успешного выполнения для объекта `p1` возвращает `p2` — объект `Promise`, находящийся

в состоянии «выполнено». Второй обработчик успешного выполнения вызывается по той простой причине, что `p2` уже успешно выполнен. Если бы `p2` потерпел отказ, вместо второго обработчика успешного выполнения был бы вызван второй обработчик отказа (если имеется).

В данном шаблоне важно понять, что второй обработчик успешного выполнения добавляется не в `p2`, а в третий объект `Promise`. Поэтому второй обработчик успешного выполнения подключается к третьему объекту `Promise`. В результате предыдущий пример можно выразить так:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
  resolve(43);
});

let p3 = p1.then(function(value) {
  // первый обработчик успешного выполнения
  console.log(value);    // 42
  return p2;
});

p3.then(function(value) {
  // второй обработчик успешного выполнения
  console.log(value);    // 43
});
```

Этот пример ясно показывает, что второй обработчик успешного выполнения подключается к `p3`, а не к `p2`. Это тонкое, но важное отличие, потому что второй обработчик успешного выполнения не будет вызван, если в `p2` произойдет отказ. Например:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
  reject(43);
});

p1.then(function(value) {
  // первый обработчик успешного выполнения
  console.log(value);    // 42
  return p2;
}).then(function(value) {
  // второй обработчик успешного выполнения
  console.log(value);    // не будет вызван
});
```

В этом примере второй обработчик успешного выполнения не будет вызван, потому что `p2` потерпел неудачу. Однако можно было бы подключить обработчик отказа:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});
```

```
let p2 = new Promise(function(resolve, reject) {
  reject(43);
});

p1.then(function(value) {
  // первый обработчик успешного выполнения
  console.log(value); // 42
  return p2;
}), catch(function(value) {
  // обработчик отказа
  console.log(value); // 43
});
```

Теперь в случае ошибки в `p2` вызывается обработчик отказа, и ему передается значение отказа 43 из `p2`.

Возврат `thenable`-объектов из обработчиков успешного или неуспешного выполнения не влияет на момент вызова исполнителя. Первый объект `Promise` в цепочке вызовет своего исполнителя первым, затем второй объект вызовет своего исполнителя и т. д. Возможность вернуть `thenable`-объект просто позволяет определить дополнительные обработчики результатов. Создавая новый объект `Promise` в обработчике успешного выполнения, вы откладываете вызов других обработчиков успешного выполнения. Например:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

p1.then(function(value) {
  console.log(value); // 42

  // Создать новый объект Promise
  let p2 = new Promise(function(resolve, reject) {
    resolve(43);
  });

  return p2
}).then(function(value) {
  console.log(value); // 43
});
```

Здесь в обработчике успешного выполнения для `p1` создается новый объект `Promise`. Это означает, что второй обработчик успешного выполнения не будет вызван, пока не выполнится `p2`. Этот шаблон удобно использовать, когда требуется дождаться выполнения предыдущей асинхронной операции, перед тем как запустить следующую.

Обработка сразу нескольких объектов Promise

Во всех примерах выше объекты `Promise` обрабатывались по одному. Но иногда желательно следить за выполнением сразу нескольких асинхронных операций (объектов `Promise`), чтобы определить следующее действие. Спецификация ECMAScript 6 определяет два метода для мониторинга нескольких объектов `Promise`: `Promise.all()` и `Promise.race()`.

Метод `Promise.all()`

Метод `Promise.all()` принимает единственный аргумент — итерируемый объект (например, массив) с объектами `Promise` для мониторинга и возвращает объект `Promise`, который выходит из состояния ожидания, только когда все `Promise` в итерируемом объекте установятся. Возвращаемый объект `Promise` переходит в состояние «выполнено», только если в этом состоянии окажутся все `Promise` в итерируемом объекте, как показано ниже:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
  resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
  resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);

p4.then(function(value) {
  console.log(Array.isArray(value)); // true
  console.log(value[0]);             // 42
  console.log(value[1]);             // 43
  console.log(value[2]);             // 44
});
```

Каждый объект `Promise` завершается успехом со своим числом. Вызов `Promise.all()` создает объект `Promise` `p4`, который переходит в состояние «выполнено», только когда все объекты — `p1`, `p2` и `p3` — перейдут в состояние «выполнено». Обработчик успешного завершения для `p4` получит массив значений успешного выполнения указанных объектов: 42, 43 и 44. Значения в массиве следуют в том же порядке, в каком были переданы объекты в вызов `Promise.all()`, благодаря чему можно установить связь между объектами `Promise` и результатами.

Если какой-либо из объектов `Promise`, переданных в `Promise.all()`, потерпит неудачу, возвращаемый объект `Promise` немедленно перейдет в состояние «отклонено», не дожидаясь завершения операций в других объектах `Promise`:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = new Promise(function(resolve, reject) {
  reject(43);
});

let p3 = new Promise(function(resolve, reject) {
  resolve(44);
});

let p4 = Promise.all([p1, p2, p3]);
```

```
p4.catch(function(value) {
    console.log(Array.isArray(value))    // false
    console.log(value);                  // 43
});
```

В этом примере `p2` отклоняется со значением 43. Обработчик отказа для `p4` вызывается немедленно, не дожидаясь, пока `p1` или `p3` завершит выполнение своей операции. (Они оба выполняют свои операции, но `p4` не будет их ждать.)

Обработчик отказа всегда получает единственное значение, а не массив, и этим значением является значение отказа из объекта `Promise`, потерпевшего неудачу. В данном случае обработчик отказа получит число 43, сообщаящее, что отказ произошел в `p2`.

Метод `Promise.race()`

Метод `Promise.race()` реализует несколько иной подход к мониторингу сразу нескольких объектов `Promise`. Он также принимает итерируемый объект с объектами `Promise` для мониторинга и возвращает `Promise`, но возвращаемый `Promise` устанавливается сразу же, как только устанавливается первый из переданных объектов `Promise`. В отличие от `Promise.all()`, метод `Promise.race()` не ждет, когда выполнятся все `Promise`, а возвращает соответствующий объект `Promise`, как только любой из переданных объектов завершит выполнение. Например:

```
let p1 = Promise.resolve(42);

let p2 = new Promise(function(resolve, reject) {
    resolve(43);
});

let p3 = new Promise(function(resolve, reject) {
    resolve(44);
});

let p4 = Promise.race([p1, p2, p3]);

p4.then(function(value) {
    console.log(value);    // 42
});
```

В этом примере `p1` создается как установившийся объект `Promise`, тогда как другие лишь планируют свои задания. Обработчик успешного завершения для `p4` вызывается немедленно со значением 42, при этом другие объекты `Promise` игнорируются. Метод `Promise.race()` проверяет переданные ему объекты `Promise`, пока не найдет первый установившийся. Если первый установившийся объект `Promise` находится в состоянии «выполнено», возвращаемый объект `Promise` также получает состояние «выполнено»; если первый установившийся объект `Promise` находится в состоянии «отклонено», возвращаемый объект `Promise` получает состояние «отклонено». Ниже приводится пример с отказом:

```
let p1 = new Promise(function(resolve, reject) {
    resolve(42);
});

let p2 = Promise.reject(43);

let p3 = new Promise(function(resolve, reject) {
```

```
    resolve(44);
  });

let p4 = Promise.race([p1, p2, p3]);

p4.catch(function(value) {
  console.log(value);    // 43
});
```

Здесь объект `p4` получит состояние «отклонено», потому что `p2` уже находится в этом состоянии к моменту вызова `Promise.race()`. Несмотря на то что `p1` и `p3` выполнились успешно, их результаты игнорируются, потому что они становятся доступны только после того, как `p2` потерпит неудачу.

Наследование Promise

`Promise` можно использовать в качестве основы для создания производных классов, как любые другие встроенные типы. Это позволяет определять свои реализации асинхронных операций, расширяющие возможности встроенного объекта `Promise`. Например, представьте, что требуется создать объект `Promise`, который может использовать методы с именами `success()` и `failure()` помимо обычных `then()` и `catch()`. Такой тип можно определить, как показано ниже:

```
class MyPromise extends Promise {

  // использовать конструктор по умолчанию

  success(resolve, reject) {
    return this.then(resolve, reject);
  }

  failure(reject) {
    return this.catch(reject);
  }
}

let promise = new MyPromise(function(resolve, reject) {
  resolve(42);
});

promise.success(function(value) {
  console.log(value);    // 42
}).failure(function(value) {
  console.log(value);
});
```

В этом примере класс `MyPromise` наследует тип `Promise` и добавляет два метода. Метод `success()` имитирует `resolve()`, а `failure()` имитирует метод `reject()`.

Оба метода используют ссылку `this` для вызова имитируемого метода. Производный класс действует точно так же, как встроенный тип `Promise`, но дополнительно теперь можно при желании вызывать методы `success()` и `failure()`.

Поскольку статические методы наследуются, производный класс также получает методы `MyPromise.resolve()`, `MyPromise.reject()`, `MyPromise.race()` и `MyPromise.all()`. Последние два метода действуют точно так же, как встроенные методы, но первые два обретают несколько иное поведение.

Оба метода, `MyPromise.resolve()` и `MyPromise.reject()`, будут возвращать экземпляры `MyPromise` независимо от передаваемых им значений, потому что для определения типа возвращаемого объекта они используют свойство `Symbol.species` (см. раздел «Свойство `Symbol.species`» в главе 9). Если в любой из методов передать встроенный объект `Promise`, он будет выполнен — успешно или неуспешно, — и метод вернет новый экземпляр `MyPromise`, поэтому вы сможете использовать новые методы для обработки успешного или неуспешного выполнения. Например:

```
let p1 = new Promise(function(resolve, reject) {
  resolve(42);
});

let p2 = MyPromise.resolve(p1);
p2.success(function(value) {
  console.log(value);           // 42
});

console.log(p2 instanceof MyPromise); // true
```

Здесь в вызов метода `MyPromise.resolve()` передается `p1` — встроенный объект `Promise`. Получившийся в результате объект `p2` является экземпляром `MyPromise`, обработчику успешного выполнения которого передается значение успешного выполнения `p1`.

Если методу `MyPromise.resolve()` или `MyPromise.reject()` передать экземпляр `MyPromise`, он просто вернет этот экземпляр, не пытаясь его выполнить. Во всех остальных отношениях эти два метода действуют точно так же, как `Promise.resolve()` и `Promise.reject()`.

Выполнение асинхронных заданий с помощью Promise

В главе 8 я познакомил вас с генераторами и показал, как можно ими пользоваться для выполнения асинхронных операций, например:

```
let fs = require("fs");

function run(taskDef) {

  // создать итератор и сделать его доступным
  // для использования в другом месте
  let task = taskDef();

  // запустить задание
  let result = task.next();

  // рекурсивная функция, продолжающая вызывать next()
  function step() {

    // если работа продолжается
    if (!result.done) {
      if (typeof result.value === "function") {
        result.value(function(err, data) {
```



```
        if (err) {
            result = task.throw(err);
            return;
        }

        result = task.next(data);
        step();
    });
} else {
    result = task.next(result.value);
    step();
}
}
// запустить обработку
step();
}

// определение функции для использования
// с инструментом запуска заданий

function readFile(filename) {
    return function(callback) {
        fs.readFile(filename, callback);
    };
}

// запустить задание

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

Эта реализация имеет несколько недостатков. Во-первых, необходимость обертывания каждой функции другой функцией, возвращающей функцию, может вызывать путаницу (даже это предложение выглядит запутанным). Во-вторых, нет никакой возможности отличить функцию, возвращающую значение, которое должно служить обратным вызовом для инструмента запуска заданий, от функции, возвращающей значение, которое не является обратным вызовом.

Этот процесс можно существенно упростить и обобщить, если каждая асинхронная операция будет возвращать объект **Promise**. Ниже приводится один из вариантов упрощенной реализации инструмента запуска заданий, использующей объекты **Promise** как обобщенный интерфейс для всего асинхронного кода:

```
let fs = require("fs");

function run(taskDef) {

    // создать итератор
    let task = taskDef();

    // запустить задание
```

```
let result = task.next();

// рекурсивная функция, выполняющая итерации
(function step() {

    // если работа продолжается
    if (!result.done) {

        // создать установившийся объект Promise
        let promise = Promise.resolve(result.value);
        promise.then(function(value) {
            result = task.next(value);
            step();
        }).catch(function(error) {
            result = task.throw(error);
            step();
        });
    }
})();

// определение функции для использования
// с инструментом запуска заданий

function readFile(filename) {
    return new Promise(function(resolve, reject) {
        fs.readFile(filename, function(err, contents) {
            if (err) {
                reject(err);
            } else {
                resolve(contents);
            }
        });
    });
}

// запустить задание

run(function*() {
    let contents = yield readFile("config.json");
    doSomethingWith(contents);
    console.log("Done");
});
```

В этой версии обобщенная функция `run()` вызывает генератор, чтобы создать итератор. Затем она вызывает `task.next()`, чтобы запустить задание, и рекурсивно продолжает вызывать `step()`, пока итератор не исчерпается.

Внутри функции `step()` свойство `result.done` имеет значение `false`, если работа еще не закончена. В этот момент свойство `result.value` должно ссылаться на объект `Promise`. На всякий случай, если вдруг используемая функция вернет что-то отличное от объекта `Promise`, вызывается `Promise.resolve()`. (Напомню, что если методу `Promise.resolve()` передать объект `Promise`, он просто вернет его, ничего не изменив, но любое другое значение завернет в объект `Promise`.) Далее добавляется обработчик успешного завершения, извлекающий значение из объекта

`Promise` и передающий это значение обратно в итератор. Затем переменной `result` присваивается следующий результат, возвращаемый итератором, после чего функция `step()` вызывает сама себя.

Обработчик отказа сохраняет любое значение отказа в объект ошибки. Метод `task.throw()` передает этот объект ошибки обратно в итератор, и если ошибка перехватывается заданием, результат присваивается следующему возвращаемому значению. В заключение внутри `catch()` вызывается `step()`, чтобы продолжить работу.

Данная функция `run()` может вызывать любые генераторы, использующие `yield` для доступа к асинхронному коду без передачи объектов `Promise` (или обратных вызовов) разработчику. В действительности, поскольку возвращаемое значение функции всегда преобразуется в объект `Promise`, эта функция может возвращать все, что угодно, а не только объекты `Promise`. Это означает, что при вызове в инструкции `yield` все методы, синхронные и асинхронные, будут работать корректно, и вам никогда не придется проверять, является ли возвращаемое ими значение объектом `Promise`.

Единственное, что от вас потребуется, — это гарантировать, что асинхронные функции, такие как `readFile()`, будут возвращать объект `Promise`, который корректно идентифицирует свое состояние. Для Node.js это означает, что вам придется преобразовать встроенные методы так, чтобы они возвращали объекты `Promise` вместо использования обратных вызовов.

ВЫПОЛНЕНИЕ АСИНХРОННЫХ ЗАДАНИЙ В БУДУЩЕМ

В настоящее время разрабатывается новый синтаксис упрощенного запуска асинхронных заданий. Например, появится новое ключевое слово `await`, своим действием близко напоминающее пример на основе объектов `Promise`, представленный выше. Основная идея заключается в использовании функции с меткой `async` вместо генератора и ключевого слова `await` вместо `yield` для вызова функции, например:

```
async function() {
  let contents = await readFile("config.json");
  doSomethingWith(contents);
  console.log("Done");
};
```

Ключевое слово `async` перед `function` указывает, что функция выполняется асинхронно. Ключевое слово `await` сигнализирует, что вызов функции `readFile("config.json")` должен вернуть объект `Promise`, а если это не так, то результат должен быть завернут в объект `Promise`. Точно так же, как функция `run()` из примера выше, `await` возбудит ошибку, если объект `Promise` получит состояние «отклонено», в противном случае вернет значение из этого объекта. В результате этих нововведений вы сможете писать асинхронный код, напоминающий синхронный и не имеющий накладных расходов на управление итератором.

Ожидается, что реализация `await` будет завершена в ECMAScript 2017 (ECMAScript 8).

В заключение

Объекты `Promise` помогают упростить асинхронное программирование на JavaScript, давая более полный контроль над асинхронными операциями и возможностью их сочетания, чем позволяют события и обратные вызовы. Объекты `Promise` планируют задания, добавляя их в очередь заданий движка JavaScript для выполнения в будущем, а вторая очередь заданий следит за состоянием объектов `Promise` и гарантирует надлежащее выполнение обработчиков успешного и неуспешного выполнения операции.

Объекты `Promise` имеют три возможных состояния: «ожидание», «выполнено» и «отклонено». Сразу после создания объект `Promise` находится в состоянии «ожидание» и переходит в состояние «выполнено» или «отклонено» в случае успешного или неуспешного завершения исполнителя. В лю-

бом случае вы можете добавлять обработчики, чтобы перехватить момент, когда объект **Promise** установится. Метод **then()** позволяет установить два обработчика, успешного и неуспешного выполнения операции, а метод **catch()** — только обработчик отказа.

Объекты **Promise** можно объединять в цепочки разными способами и передавать информацию между ними. Каждый вызов **then()** создает и возвращает новый объект **Promise**, который установится, когда установится предшествующий объект. Такие цепочки можно использовать для реализации реакции на последовательность асинхронных событий. Для мониторинга и обработки нескольких объектов **Promise** можно также использовать методы **Promise.race()** и **Promise.all()**.

Запуск асинхронных заданий реализуется проще, когда объекты **Promise** используются в комбинации с генераторами, потому что объекты **Promise** имеют обобщенный интерфейс, который могут возвращать асинхронные операции. Благодаря этому можно использовать генераторы и оператор **yield**, чтобы дождаться завершения асинхронной операции и обработать ее результат.

Многие новые библиотеки для веб-приложений основаны на объектах **Promise**, и вы можете быть уверены, что в будущем появятся и другие библиотеки, следующие их примеру.

Прокси-объекты и Reflection API

Одной из целей обеих спецификаций, ECMAScript 5 и ECMAScript 6, было снятие завесы таинственности с функциональных возможностей JavaScript. Например, до ECMAScript 5 встроенные объекты JavaScript могли иметь свойства, перечислимые и недоступные для записи, но разработчики не имели возможности определять собственные свойства как перечислимые или недоступные для записи. В ECMAScript 5 появился метод `Object.defineProperty()`, который позволил разработчикам делать то, что движки JavaScript давно уже умели делать.

Спецификация ECMAScript 6 дает разработчикам еще более полный доступ к возможностям движка JavaScript, добавляя встроенные объекты. Чтобы позволить разработчикам создавать встроенные объекты, язык открывает доступ к внутренним механизмам объектов посредством *прокси-объектов (proxies)* — оберток, которые могут перехватывать и изменять порядок выполнения низкоуровневых операций движком JavaScript. В этой главе сначала подробно описывается проблема, решить которую призваны прокси-объекты, а затем обсуждаются приемы создания и эффективного использования прокси-объектов.

Проблема с массивами

До выхода ECMAScript 6 разработчики не могли имитировать работу объектов массивов в собственных объектах. Свойство `length` массива изменяется автоматически, когда программа присваивает значения определенным элементам этого массива, а изменяя свойство `length`, можно оказывать влияние на элементы массива. Например:

```
let colors = ["red", "green", "blue"];

console.log(colors.length);    // 3

colors[3] = "black";

console.log(colors.length);    // 4

console.log(colors[3]);        // "black"

colors.length = 2;

console.log(colors.length);    // 2
console.log(colors[3]);        // undefined
```

```
console.log(colors[2]);      // undefined
console.log(colors[1]);      // "green"
```

Первоначально массив `colors` имеет три элемента. Операция присваивания строки `"black"` элементу `colors[3]` автоматически увеличивает свойство `length` до 4. Присваивание числа 2 свойству `length` удаляет два последних элемента из массива, оставляя два первых. В ECMAScript 5 отсутствовали механизмы, которые позволили бы разработчикам реализовать аналогичное поведение, но прокси-объекты изменили ситуацию.

ПРИМЕЧАНИЕ

Такое нестандартное поведение свойств с числовыми именами и свойства `length` объясняет, почему в ECMAScript 6 массивы считаются экзотическими объектами.

Введение в прокси-объекты и Reflection API

Вызов `new Proxy()` создает прокси-объект для использования вместо другого объекта (называется *целевым*). Прокси *виртуализирует* цель так, что оба объекта, прокси и целевой, выглядят функционально одинаковыми.

Прокси-объекты позволяют перехватывать низкоуровневые операции целевого объекта, которые иначе являются внутренними для движка JavaScript. Перехват этих низкоуровневых операций производится с помощью *ловушки* — функции, отвечающей за определенную операцию.

Программный интерфейс механизма рефлексии (Reflection API) представлен объектом `Reflect` — коллекцией методов, которые предоставляют поведение по умолчанию к тем же низкоуровневым операциям, которые могут переопределять прокси-объекты. Каждой ловушке в прокси соответствует метод в `Reflect`. Эти методы имеют те же имена и принимают те же аргументы, что и соответствующие им ловушки в прокси. В табл. 12.1 приводится краткая сводка по ловушкам в прокси.

Каждая ловушка переопределяет некоторую встроенную операцию с объектами JavaScript, позволяя вам перехватывать и изменять их. Если понадобится использовать неизменные встроенные операции, они всегда доступны в виде соответствующих методов Reflection API. Связь между прокси-объектами и Reflection API станет понятнее, когда вы начнете создавать прокси, поэтому продолжайте читать и исследуйте предлагаемые примеры.

Таблица 12.1. Прокси-ловушки в JavaScript

Ловушка в прокси	Переопределяемая операция	Метод по умолчанию
<code>get</code>	Чтение значения свойства	<code>Reflect.get()</code>
<code>set</code>	Запись значения в свойство	<code>Reflect.set()</code>
<code>has</code>	Оператор <code>in</code>	<code>Reflect.has()</code>
<code>deleteProperty</code>	Оператор <code>delete</code>	<code>Reflect.deleteProperty()</code>
<code>getPrototypeOf</code>	<code>Object.getPrototypeOf()</code>	<code>Reflect.getPrototypeOf</code>
<code>setPrototypeOf</code>	<code>Object.setPrototypeOf()</code>	<code>Reflect.setPrototypeOf()</code>
<code>isExtensible</code>	<code>Object.isExtensible()</code>	<code>Reflect.isExtensible()</code>
<code>preventExtensions</code>	<code>Object.preventExtensions()</code>	<code>Reflect.preventExtensions()</code>
<code>getOwnPropertyDescriptor</code>	<code>Object.getOwnPropertyDescriptor()</code>	<code>Reflect.getOwnPropertyDescriptor()</code>
<code>defineProperty</code>	<code>Object.defineProperty()</code>	<code>Reflect.defineProperty</code>
<code>ownKeys</code>	<code>Object.keys()</code> , <code>Object.getOwnPropertyNames()</code> и <code>Object.getOwnPropertySymbols()</code>	<code>Reflect.ownKeys()</code>
<code>apply</code>	Вызов функции	<code>Reflect.apply()</code>
<code>construct</code>	Вызов функции с ключевым словом <code>new</code>	<code>Reflect.construct()</code>

ПРИМЕЧАНИЕ

В первоначальной версии спецификации ECMAScript 6 имелась еще одна ловушка — `enumerate`, предназначенная для изменения поведения перечисления свойств объекта в цикле `for-in` и методе `Object.keys()`. Однако она была убрана в спецификации ECMAScript 7 (ECMAScript 2016), потому что в ходе реализации вскрылись некоторые сложности. Ловушка `enumerate` не поддерживается ни в одном окружении JavaScript и потому не рассматривается в этой главе.

Создание простого прокси-объекта

Когда вызывается конструктор `Proxy`, чтобы создать прокси-объект, ему передаются два аргумента: цель и обработчик. *Обработчик* — это объект, определяющий одну или несколько ловушек. Прокси-объект использует реализации по умолчанию для всех операций, кроме тех, для которых определены ловушки. Чтобы создать простейший прокси-объект, который лишь передает вызовы реализации по умолчанию, можно использовать обработчик без ловушек, например:

```
let target = {};  
  
let proxy = new Proxy(target, {});  
  
proxy.name = "proxy";  
console.log(proxy.name);    // "proxy"  
console.log(target.name);   // "proxy"  
  
target.name = "target";  
console.log(proxy.name);    // "target"  
console.log(target.name);   // "target"
```

В этом примере прокси-объект `проху` делегирует выполнение всех операций непосредственно целевому объекту `target`. Когда свойству `proxy.name` присваивается значение `"proxy"`, в объекте `target` создается свойство `name`. Сам объект `проху` не хранит это свойство; он просто передает операцию объекту `target`. Аналогично обращения к `proxy.name` и `target.name` возвращают одно и то же значение, потому что оба они в конечном итоге ссылаются на свойство `target.name`. Это также означает, что если присвоить свойству `target.name` новое значение, оно станет также доступно через `proxy.name`. Конечно, прокси-объекты без ловушек не представляют особого интереса, поэтому давайте посмотрим, что получится, если определить какую-нибудь ловушку

Проверка свойств с помощью ловушки `set`

Предположим, что требуется создать объект, свойства которого могут принимать только числовые значения. Это означает необходимость проверки каждого нового свойства, добавляемого в объект, и возбуждения ошибки, если присваиваемое ему значение не является числом. Для решения этой задачи можно определить ловушку `set`, переопределяющую поведение по умолчанию операции присваивания значения. Ловушка `set` принимает четыре аргумента:

- `trapTarget`. Объект, в который добавляется свойство (цель для прокси-объекта).
- `key`. Ключ свойства (строка или символ) для записи.
- `value`. Значение, записываемое в свойство.
- `receiver`. Объект, в котором определена выполняемая операция (обычно прокси-объект).

Ловушке `set` соответствует метод `Reflect.set()`, который определяет поведение по умолчанию данной операции. Метод `Reflect.set()` принимает те же четыре аргумента, что и ловушка `set`, что дает возможность использовать этот метод внутри ловушки. Ловушка должна вернуть `true`, если значение было присвоено свойству, и `false` — в противном случае. (Метод `Reflect.set()` возвращает правильное значение в зависимости от успеха операции.)

Для проверки значений, присваиваемых свойствам, можно использовать ловушку `set`. Например:

```
let target = {
  name: "target"
};

let proxy = new Proxy(target, {
  set(trapTarget, key, value, receiver) {

    // игнорировать существующие свойства объекта,
    // на которые действие ловушки не распространяется
    if (!trapTarget.hasOwnProperty(key)) {
      if (isNaN(value)) {
        throw new TypeError("Property must be a number.");
      }
    }

    // добавить свойство
    return Reflect.set(trapTarget, key, value, receiver);
  }
});

// добавление нового свойства
proxy.count = 1;
console.log(proxy.count);    // 1
console.log(target.count);   // 1

// свойству name можно присвоить строку,
// потому что оно уже существует
proxy.name = "proxy";
console.log(proxy.name);     // "proxy"
console.log(target.name);    // "proxy"

// следующая операция вызовет ошибку
proxy.anotherName = "proxy";
```

В этом примере определяется ловушка в прокси-объекте, которая проверяет значение любого нового свойства, добавляемого в объект `target`. Когда выполняется инструкция `proxy.count = 1`, вызывается ловушка `set`. В аргументе `trapTarget` передается ссылка на `target`, в аргументе `key` — строка `"count"`, в аргументе `value` — значение `1` и в аргументе `receiver` (в этом примере не используется) — ссылка на `proxy`. В объекте `target` отсутствует свойство с именем `count`, поэтому ловушка проверяет значение `value`, передавая его в вызов `isNaN()`. Если в результате будет получено значение `NaN`, значит, производится попытка присвоить свойству нечисловое значение, и поэтому возбуждается ошибка. Так как в этом примере свойству `count` присваивается `1`, ловушка вызывает `Reflect.set()` с теми же четырьмя аргументами, которые получила сама, чтобы добавить новое свойство.

Когда свойству `proxy.name` присваивается строка, операция выполняется успешно. Так как объект `target` уже имеет свойство `name`, оно исключается из проверки вызовом метода `trapTarget`

`.hasOwnProperty()`. Это гарантирует поддержку свойств, которые прежде имели нечисловые значения.

Однако попытка присвоить строку свойству `proxy.anotherName` вызывает ошибку. Свойство `anotherName` не существовало в объекте `target` прежде, поэтому оно подвергается проверке. В процессе проверки значения возбуждается ошибка, потому что `"proxy"` не является числовым значением.

Ловушка `set` позволяет перехватывать операции записи в свойства, а ловушка `get` — операции чтения свойств.

Проверка формы объектов с помощью ловушки `get`

Одной из необычных особенностей JavaScript, которая иногда может вводить в заблуждение, является операция чтения свойств, которая не вызывает ошибку, если предпринимается попытка прочесть несуществующее свойство. Вместо ошибки она просто возвращает значение `undefined`, как в следующем примере:

```
let target = {};  
  
console.log(target.name);    // undefined
```

В большинстве других языков попытка прочесть `target.name` вызовет ошибку, потому что свойство не существует. Но в JavaScript в качестве значения свойства `target.name` будет возвращено `undefined`. Если вам доводилось работать с большими программами, вы наверняка знаете, что подобное поведение может вызывать существенные проблемы, особенно когда в имени свойства допущена опечатка. Прокси-объекты помогают решить эту проблему за счет организации проверки формы объекта.

Форма объекта (object shape) — это коллекция свойств и методов, доступных в объекте. Движки JavaScript используют формы объектов для оптимизации кода, часто создавая классы, представляющие объекты. Если можно с уверенностью сказать, что объект всегда будет иметь один и тот же набор свойств и методов (такое поведение можно гарантировать с помощью методов `Object.preventExtensions()`, `Object.seal()` или `Object.freeze()`), тогда возбуждение ошибки при попытке обратиться к несуществующему свойству может оказаться полезным приобретением. Прокси-объекты позволяют легко проверить форму объекта.

Так как проверка свойства должна выполняться только при чтении свойств, ее можно реализовать в ловушке `get`. Ловушка `get` вызывается, когда производится операция чтения свойства, даже если это свойство отсутствует в объекте. Она принимает три аргумента:

- `trapTarget`. Объект, свойство которого читается (цель для прокси-объекта),
- `key`. Ключ свойства (строка или символ) для чтения.
- `receiver`. Объект, в котором определена выполняемая операция (обычно прокси-объект).

Эти аргументы повторяют аргументы ловушки `set` с одним заметным исключением: отсутствует аргумент `value`, потому что ловушка `get` не выполняет запись значения. Ловушке `get` соответствует метод `Reflect.get()`, который принимает те же три аргумента, что и ловушка `get`, и возвращает значение свойства.

Ловушку `get` и метод `Reflect.get()` можно использовать, чтобы возбуждать ошибку при любой попытке прочесть свойство, отсутствующее в целевом объекте, как показано ниже:

```
let proxy = new Proxy({}, {  
  get(trapTarget, key, receiver) {  
    if (!(key in receiver)) {
```

```
        throw new TypeError("Property " + key + " doesn't exist.");
    }

    return Reflect.get(trapTarget, key, receiver);
}

});

// допускается добавление свойств
proxy.name = "проху";
console.log(proxy.name);    // "проху"

// обращение к несуществующему свойству вызывает ошибку
console.log(proxy.nme);    // throws an error
```

Ловушка `get` в этом примере перехватывает операции чтения свойств. Оператор `in` проверяет наличие свойства в объекте `receiver`. Здесь в операторе `in` используется `receiver` вместо `trapTarget` на тот случай, если в прокси-объекте `receiver` определена ловушка `has`, о которой я расскажу в следующем разделе. Использование `trapTarget` в такой ситуации приведет к вызову ловушки `has` и теоретически может дать неверный результат. Если свойство отсутствует, возбуждается ошибка; в противном случае используется поведение по умолчанию.

Это решение позволяет добавлять новые свойства, такие как `proxy.name`, записывать значения и читать их. Последняя строка в примере содержит опечатку: под `proxy.nme` программист, вероятно, подразумевал `proxy.name`. Она вызовет ошибку, потому что свойство `nme` отсутствует в объекте.

Соккрытие свойств с помощью ловушки `has`

Оператор `in` проверяет наличие свойства в указанном объекте и возвращает `true`, если свойство, соответствующее заданному имени или символу, имеется. Например:

```
let target = {
  value: 42;
}

console.log("value" in target);    // true
console.log("toString" in target); // true
```

Объект имеет оба свойства, `value` и `toString`, поэтому в обоих случаях оператор `in` вернул `true`. Свойство `value` — это собственное свойство, тогда как свойство `toString` принадлежит прототипу (унаследовано от `Object`). Прокси-объекты позволяют перехватить эту операцию с помощью ловушки `has` и вернуть другое значение.

Ловушка `has` вызывается при любом использовании оператора `in`. Она принимает два аргумента:

- `trapTarget`. Объект, в котором определяется присутствие свойства (цель для прокси-объекта).
- `key`. Ключ свойства (строка или символ) для проверки.

Метод `Reflect.has()` принимает те же два аргумента и реализует поведение по умолчанию оператора `in`. Ловушку `has` и метод `Reflect.has()` можно применить, чтобы изменить поведение оператора `in` в отношении одних свойств и использовать поведение по умолчанию в отношении других. Например, можно скрыть свойство `value` из предыдущего примера:

```
let target = {
  name: "target",
  value: 42
};

let proxy = new Proxy(target, {
  has(trapTarget, key) {
    if (key === "value") {
      return false;
    } else {
      return Reflect.has(trapTarget, key);
    }
  }
});

console.log("value" in proxy);      // false
console.log("name" in proxy);      // true
console.log("toString" in proxy);  // true
```

Ловушка `has` в объекте `proxy` сравнивает аргумент `key` со строкой `"value"` и возвращает `false`, если условие выполняется. В противном случае вызывается метод `Reflect.has()`, реализующий поведение по умолчанию. В результате для свойства `value` оператор `in` возвращает `false`, даже если оно существует в объекте `target`. Для других свойств, `name` и `toString`, корректно возвращается `true`, когда их наличие проверяется оператором `in`.

Предотвращение удаления свойств с помощью ловушки `deleteProperty`

Оператор `delete` удаляет свойство из объекта и возвращает `true`, если удаление прошло успешно, и `false` — в противном случае. При попытке удалить ненастраиваемое свойство в строгом режиме `delete` возбуждает ошибку, а в обычном режиме `delete` просто возвращает `false`. Например:

```
let target = {
  name: "target",
  value: 42
};

Object.defineProperty(target, "name", { configurable: false });

console.log("value" in target);    // true

let result1 = delete target.value;
console.log(result1);              // true

console.log("value" in target);    // false
// примечание: следующая строка вызовет ошибку в строгом режиме
let result2 = delete target.name;
console.log(result2);              // false

console.log("name" in target);     // true
```

Свойство `value` удаляется с помощью оператора `delete`, и в результате оператор `in` возвращает `false` в третьем вызове `console.log()`. Ненастраиваемое свойство `name` нельзя удалить, поэтому оператор `delete` просто возвращает `false` (если выполнить этот код в строгом режиме, он вызовет ошибку). Такое поведение можно изменить с помощью ловушки `deleteProperty` в прокси-объекте.

Всякий раз когда к свойству объекта применяется оператор `delete`, вызывается ловушка `deleteProperty`. Она получает два аргумента:

- `trapTarget`. Объект, из которого удаляется свойство (цель для прокси-объекта).
- `key`. Ключ свойства (строка или символ) для удаления.

Метод `Reflect.deleteProperty()` принимает те же два аргумента и реализует поведение по умолчанию оператора `delete`. Ловушку `deleteProperty` и метод `Reflect.deleteProperty()` можно использовать, чтобы изменить поведение оператора `delete`. Например, можно предотвратить удаление свойства `value`:

```
let target = {
  name: "target",
  value: 42
};

let proxy = new Proxy(target, {
  deleteProperty(trapTarget, key) {

    if (key === "value") {
      return false;
    } else {
      return Reflect.deleteProperty(trapTarget, key);
    }
  }
});

// попытаться удалить свойство proxy.value

console.log("value" in proxy); // true

let result1 = delete proxy.value;
console.log(result1);          // false

console.log("value" in proxy); // true

// попытаться удалить свойство proxy.name

console.log("name" in proxy);  // true

let result2 = delete proxy.name;
console.log(result2);          // true

console.log("name" in proxy);  // false
```

Этот пример очень похож на пример реализации ловушки `has`. Ловушка `deleteProperty` сравнивает аргумент `key` со строкой `"value"` и возвращает `false`, если условие выполняется. В противном случае вызывается метод `Reflect.deleteProperty()`.

Свойство `value` нельзя удалить через `proxy`, потому что эта операция перехватывается, но свойство `name` удаляется без осложнений. Данный прием может пригодиться, когда требуется защитить свойства от удаления и не вызвать ошибку в строгом режиме.

Ловушки операций с прототипом

В главе 4 был представлен метод `Object.setPrototypeOf()`, добавленный спецификацией ECMAScript 6 в дополнение к методу `Object.getPrototypeOf()`, который появился в ECMAScript 5. Прокси-объекты позволяют перехватывать вызовы обоих методов с помощью ловушек `setPrototypeOf` и `getPrototypeOf`. В обоих случаях метод в `Object` вызывает соответствующую ловушку в прокси-объекте, давая возможность изменить поведение метода.

Поскольку эти две ловушки связаны с прототипами прокси-объектов, каждой из них соответствует несколько методов. Ловушка `setPrototypeOf` принимает следующие аргументы:

- `trapTarget`. Объект, для которого устанавливается прототип (цель для прокси-объекта).
- `proto`. Объект для использования в роли прототипа.

Те же аргументы передаются в методы `Object.setPrototypeOf()` и `Reflect.setPrototypeOf()`. Ловушка `getPrototypeOf` принимает только аргумент `trapTarget`, который также передается в методы `Object.getPrototypeOf()` и `Reflect.getPrototypeOf()`.

Как действуют ловушки операций с прототипом

Ловушки операций с прототипом имеют некоторые ограничения. Во-первых, ловушка `getPrototypeOf` должна возвращать объект или `null` — возврат любого другого значения вызывает ошибку во время выполнения. Проверка возвращаемого значения гарантирует, что `Object.getPrototypeOf()` всегда будет возвращать ожидаемое значение. Во-вторых, ловушка `setPrototypeOf` должна возвращать `false`, если операция потерпела неудачу. Когда `setPrototypeOf` возвращает `false`, `Object.setPrototypeOf()` возбуждает ошибку. Если `setPrototypeOf` возвращает любое другое значение, отличное от `false`, метод `Object.setPrototypeOf()` предполагает, что операция преуспела.

Следующий пример скрывает прототип прокси-объекта, всегда возвращая `null`, а также не позволяет изменять прототип:

```
let target = {};  
let proxy = new Proxy(target, {  
  getPrototypeOf(trapTarget) {  
    return null;  
  },  
  
  setPrototypeOf(trapTarget, proto) {  
    return false;  
  }  
});  
  
let targetProto = Object.getPrototypeOf(target);  
let proxyProto = Object.getPrototypeOf(proxy);  
  
console.log(targetProto === Object.prototype); // true  
console.log(proxyProto === Object.prototype);  // false  
console.log(proxyProto);                       // null
```

```
// завершится успехом
Object.setPrototypeOf(target, {});

// вызовет ошибку
Object.setPrototypeOf(proxy, {});
```

Этот пример подчеркивает разницу между поведением `target` и `proxy`. Для `target` метод `Object.getPrototypeOf()` возвращает его прототип, а для `proxy` возвращается `null`, потому что вызывается ловушка `getPrototypeOf`. Аналогично метод `Object.setPrototypeOf()` завершается успехом, когда он вызывается для объекта `target`, но возбуждает ошибку, когда вызывается для `proxy`, из-за вмешательства ловушки `setPrototypeOf`.

Если вдруг понадобится поведение по умолчанию для этих двух ловушек, используйте соответствующие методы объекта `Reflect`. Например, следующий код реализует поведение по умолчанию для ловушек `getPrototypeOf` и `setPrototypeOf`:

```
let target = {};
let proxy = new Proxy(target, {
  getPrototypeOf(trapTarget) {
    return Reflect.getPrototypeOf(trapTarget);
  },

  setPrototypeOf(trapTarget, proto) {
    return Reflect.setPrototypeOf(trapTarget, proto);
  }
});

let targetProto = Object.getPrototypeOf(target);
let proxyProto = Object.getPrototypeOf(proxy);

console.log(targetProto === Object.prototype); // true
console.log(proxyProto === Object.prototype);  // true

// завершится успехом
Object.setPrototypeOf(target, {});

// также завершится успехом
Object.setPrototypeOf(proxy, {});
```

В этом примере `target` и `proxy` можно использовать взаимозаменяемо и получать одинаковые результаты, потому что ловушки `getPrototypeOf` и `setPrototypeOf` просто вызывают реализацию по умолчанию. Необходимо отметить, что из-за некоторых важных различий в этом примере используются методы `Reflect.getPrototypeOf()` и `Reflect.setPrototypeOf()` вместо одноименных методов объекта `Object`.

Почему поддерживается два набора методов?

Подозрительное сходство методов `Reflect.getPrototypeOf()` и `Reflect.setPrototypeOf()` и методов `Object.getPrototypeOf()` и `Object.setPrototypeOf()` часто вызывает путаницу. Несмотря на то что оба набора методов выполняют схожие операции, между ними все же есть заметные различия.

Методы `Object.getPrototypeOf()` и `Object.setPrototypeOf()` — операции более высокого уровня, изначально создававшиеся для разработчиков, а методы `Reflect.getPrototypeOf()`

и `Reflect.setPrototypeOf()` — низкоуровневые операции, открывающие доступ к первоначальным, внутренним операциям `[[GetPrototypeOf]]` и `[[SetPrototypeOf]]`. Метод `Reflect.getPrototypeOf()` — это обертка для внутренней операции `[[GetPrototypeOf]]` (с дополнительной проверкой входных данных). Методы `Reflect.setPrototypeOf()` и `[[SetPrototypeOf]]` имеют такое же родство. Соответствующие методы в `Object` также вызывают `[[GetPrototypeOf]]` и `[[SetPrototypeOf]]`, но перед этим выполняют несколько дополнительных шагов и проверяют возвращаемое значение, чтобы определить, как действовать дальше.

Метод `Reflect.getPrototypeOf()` возбуждает ошибку, если его аргумент не является объектом, а `Object.getPrototypeOf()` сначала приводит значение к типу объекта и только потом выполняет операцию. Если передать этим методам число, они вернут разные результаты:

```
let result1 = Object.getPrototypeOf(1);
console.log(result1 === Number.prototype);    // true

// вызовет ошибку
Reflect.getPrototypeOf(1);
```

Метод `Object.getPrototypeOf()` позволяет получить прототип для числа 1, потому что сначала он приводит значение к типу `Number`, а затем возвращает `Number.prototype`. Метод `Reflect.getPrototypeOf()` не выполняет такого приведения, а так как 1 не является объектом, возбуждает ошибку.

Метод `Reflect.setPrototypeOf()` также действует иначе, чем метод `Object.setPrototypeOf()`. В частности, `Reflect.setPrototypeOf()` возвращает логическое значение, сообщаемое об успехе операции. В случае успеха возвращается значение `true`, а в случае неудачи — `false`. Если `Object.setPrototypeOf()` терпит неудачу, он возбуждает ошибку.

Как показывает первый пример в разделе «Как действуют ловушки операций с прототипом», когда ловушка `setPrototypeOf` возвращает `false`, метод `Object.setPrototypeOf()` возбуждает ошибку. Метод `Object.setPrototypeOf()` возвращает первый аргумент (его значение) и потому не подходит для реализации поведения по умолчанию ловушки `setPrototypeOf`. Эти различия демонстрируются ниже:

```
let target1 = {};
let result1 = Object.setPrototypeOf(target1, {});
console.log(result1 === target1);    // true

let target2 = {};
let result2 = Reflect.setPrototypeOf(target2, );
console.log(result2 === target2);    // false
console.log(result2);                // true
```

В этом примере вызов `Object.setPrototypeOf()` возвращает `target1`, а вызов `Reflect.setPrototypeOf()` возвращает `true`. Это тонкое отличие имеет большое значение. Вы увидите еще не одну пару методов в `Object` и `Reflect`, которые на первый взгляд дублируют друг друга, но в своих ловушках всегда используйте методы объекта `Reflect`.

ПРИМЕЧАНИЕ

При использовании внутри прокси-объектов методы `Reflect.getPrototypeOf()`/`Object.getPrototypeOf()` и `Reflect.setPrototypeOf()` / `Object.setPrototypeOf()` будут вызывать ловушки `getPrototypeOf` и `setPrototypeOf` соответственно.

Ловушки, связанные с расширяемостью объектов

В ECMAScript 5 была добавлена возможность управления способностью объектов к расширению в виде пары методов `Object.preventExtensions()` и `Object.isExtensible()`, а в ECMAScript 6 появилась возможность перехватывать вызовы этих методов с помощью ловушек `preventExtensions` и `isExtensible`. Обе ловушки принимают единственный аргумент `trapTarget` — объект, для которого был вызван метод. Ловушка `isExtensible` должна вернуть логическое значение, сообщающее о способности объекта расширяться. Ловушка `preventExtensions` также должна вернуть логическое значение, сообщающее об успехе операции. Методы `Reflect.preventExtensions()` и `Reflect.isExtensible()` реализуют поведение по умолчанию. Оба возвращают логическое значение, поэтому их можно непосредственно использовать в соответствующих ловушках.

Два простых примера

Чтобы увидеть, как действуют ловушки управления расширяемостью объектов, взгляните на следующий пример, реализующий поведение по умолчанию ловушек `isExtensible` и `preventExtensions`:

```
let target = {};  
let proxy = new Proxy(target, {  
  isExtensible(trapTarget) {  
    return Reflect.isExtensible(trapTarget);  
  },  
  
  preventExtensions(trapTarget) {  
    return Reflect.preventExtensions(trapTarget);  
  }  
});  
  
console.log(Object.isExtensible(target)); // true  
console.log(Object.isExtensible(proxy));  // true  
  
Object.preventExtensions(proxy);  
  
console.log(Object.isExtensible(target)); // false  
console.log(Object.isExtensible(proxy));  // false
```

Этот пример показывает, что вызовы методов `Object.preventExtensions()` и `Object.isExtensible()` корректно проходят через `proxy` в `target`. При необходимости вы можете изменить их поведение. Например, если нежелательно, чтобы `Object.preventExtensions()` успешно воздействовал на прокси-объект, из ловушки `preventExtensions` можно вернуть `false`:

```
let target = {};  
let proxy = new Proxy(target, {  
  isExtensible(trapTarget) {  
    return Reflect.isExtensible(trapTarget);  
  },  
  
  preventExtensions(trapTarget) {  
    return false  
  }  
});
```



```
console.log(Object.isExtensible(target));    // true
console.log(Object.isExtensible(proxy));      // true

Object.preventExtensions(proxy);

console.log(Object.isExtensible(target));    // true
console.log(Object.isExtensible(proxy));      // true
```

Здесь вызов `Object.preventExtensions(proxy)` фактически игнорируется, потому что ловушка `preventExtensions` возвращает `false`. Операция не передается в объект `target`, поэтому `Object.isExtensible()` возвращает `true`.

Дубликаты методов управления расширяемостью

Возможно, вы обратили внимание, что мы опять столкнулись с повторяющимися методами в `Object` и `Reflect`. Но в данном случае они имеют больше сходства. Методы `Object.isExtensible()` и `Reflect.isExtensible()` отличаются только реакцией при получении аргумента, не являющегося объектом. В этом случае `Object.isExtensible()` всегда возвращает `false`, а `Reflect.isExtensible()` возбуждает ошибку. Это поведение демонстрирует следующий пример:

```
let result1 = Object.isExtensible(2);
console.log(result1);    // false

// вызовет ошибку
let result2 = Reflect.isExtensible(2);
```

Это напоминает различия между методами `Object.getPrototypeOf()` и `Reflect.getPrototypeOf()`: метод более низкого уровня выполняет более строгую проверку ошибок.

Методы `Object.preventExtensions()` и `Reflect.preventExtensions()` также очень похожи между собой. Метод `Object.preventExtensions()` всегда возвращает значение своего аргумента, даже если это не объект. Метод `Reflect.preventExtensions()`, напротив, возбуждает ошибку, если его аргумент не является объектом; в противном случае `Reflect.preventExtensions()` возвращает `true` в случае успеха операции и `false`, если операция завершилась неудачей. Например:

```
let result1 = Object.preventExtensions(2);
console.log(result1);    // 2

let target = {};
let result2 = Reflect.preventExtensions(target);
console.log(result2);    // true

// вызовет ошибку
let result3 = Reflect.preventExtensions(2);
```

Здесь метод `Object.preventExtensions()` возвращает полученное значение `2`, несмотря на то что `2` — это не объект. Метод `Reflect.preventExtensions()` возвращает `true`, когда получает объект, и возбуждает ошибку, когда получает `2`.

Ловушки операций с дескрипторами свойств

Одним из важнейших новшеств в ECMAScript 5 стала возможность определять атрибуты свойств с помощью метода `Object.defineProperty()`. В ранних версиях JavaScript не было никакой возможности определить методы доступа к свойствам, сделать свойство доступным только для чтения или перечислимым. Все это стало возможным с появлением метода `Object.defineProperty()`, а атрибуты можно извлекать с помощью метода `Object.getOwnPropertyDescriptor()`.

Прокси-объекты позволяют перехватывать вызовы методов `Object.defineProperty()` и `Object.getOwnPropertyDescriptor()` в ловушках `defineProperty` и `getOwnPropertyDescriptor` соответственно. Ловушка `defineProperty` принимает следующие аргументы:

- `trapTarget`. Объект, в котором определяется свойство (цель для прокси-объекта).
- `key`. Ключ свойства (строка или символ),
- `descriptor`. Объект-дескриптор для свойства.

Ловушка `defineProperty` должна возвращать `true` в случае успеха операции и `false` — в противном случае. Ловушка `getOwnPropertyDescriptor` принимает только аргументы `trapTarget` и `key` и, как нетрудно догадаться, возвращает дескриптор. Соответствующие методы `Reflect.defineProperty()` и `Reflect.getOwnPropertyDescriptor()` принимают те же аргументы, что и ловушки. Следующий пример реализует поведение по умолчанию для каждой ловушки:

```
let proxy = new Proxy({}, {
  defineProperty(trapTarget, key, descriptor) {
    return Reflect.defineProperty(trapTarget, key, descriptor);
  },

  getOwnPropertyDescriptor(trapTarget, key) {
    return Reflect.getOwnPropertyDescriptor(trapTarget, key);
  }
});

Object.defineProperty(proxy, "name", {
  value: "proxy"
});

console.log(proxy.name);           // "proxy"

let descriptor = Object.getOwnPropertyDescriptor(proxy, "name");

console.log(descriptor.value);     // "proxy"
```

Этот код определяет в объекте `proxy` свойство с именем `"name"`, используя метод `Object.defineProperty()`. Затем он извлекает дескриптор для этого свойства вызовом `Object.getOwnPropertyDescriptor()`.

Блокирование вызова `Object.defineProperty()`

Ловушка `defineProperty` должна возвращать логическое значение, определяющее успех операции. Когда возвращается значение `true`, метод `Object.defineProperty()` завершается успехом; когда возвращается `false`, метод `Object.defineProperty()` возбуждает ошибку. Это обстоятельство можно использовать для ограничения видов свойств, которые можно определить с помощью метода `Object.defineProperty()`. Например, чтобы предотвратить возможность определения символьных свойств, можно проверить тип ключа и, если он не является строкой, возвращать `false`, например:

```
let proxy = new Proxy({}, {
  defineProperty(trapTarget, key, descriptor) {

    if (typeof key === "symbol") {
      return false;
    }

    return Reflect.defineProperty(trapTarget, key, descriptor);
  }
});

Object.defineProperty(proxy, "name", {
  value: "proxy"
});

console.log(proxy.name);    // "proxy"

let nameSymbol = Symbol("name");

// вызовет ошибку
Object.defineProperty(proxy, nameSymbol, {
  value: "proxy"
});
```

Получив в аргументе `key` символ, ловушка `defineProperty` вернет `false`, в противном случае вызовет реализацию по умолчанию. Когда метод `Object.defineProperty()` вызывается с ключом `"name"`, он выполняется успешно, потому что ключ является строкой. Когда `Object.defineProperty()` вызывается с символом `nameSymbol`, он возбуждает ошибку, потому что ловушка `defineProperty` возвращает `false`.

ПРИМЕЧАНИЕ

Можно также скрыть ошибку в вызове `Object.defineProperty()`, просто возвращая `true` и не вызывая `Reflect.defineProperty()`. В этом случае ошибка возбуждаться не будет, даже при невозможности определить свойство.

Ограничения объекта дескриптора

Чтобы гарантировать единообразие поведения при использовании методов `Object.defineProperty()` и `Object.getOwnPropertyDescriptor()`, объекты дескрипторов передаются в ловушку `defineProperty` в нормализованном виде. По той же причине объекты, возвращаемые ловушкой `getOwnPropertyDescriptor`, также всегда подвергаются проверке.

Тип объекта, передаваемого методу `Object.defineProperty()` в третьем аргументе, не имеет большого значения; ловушка `defineProperty` получит объект дескриптора, обладающего только свойствами `enumerable`, `configurable`, `value`, `writable`, `get` и `set`. Например:

```
let proxy = new Proxy({}, {
  defineProperty(trapTarget, key, descriptor) {
    console.log(descriptor.value);    // "proxy"
    console.log(descriptor.name);    // undefined

    return Reflect.defineProperty(trapTarget, key, descriptor);
  }
});
```

```
});  
  
Object.defineProperty(proxy, "name", {  
  value: "proxy",  
  name: "custom"  
});
```

Здесь в третьем аргументе методу `Object.defineProperty()` передается объект с нестандартным свойством `name`. Когда вызывается ловушка `defineProperty`, она получает объект дескриптора без свойства `name`, но со свойством `value`. Причина этого в том, что `descriptor` в действительности не является простой ссылкой на третий аргумент в вызове `Object.defineProperty()`, но представляет собой новый объект, содержащий только допустимые свойства. Метод `Reflect.defineProperty()` также игнорирует любые нестандартные свойства в дескрипторе.

Ловушка `getOwnPropertyDescriptor` имеет несколько отличающееся ограничение — она должна возвращать `null`, `undefined` или объект. Объект, возвращаемый ловушкой, должен содержать только следующие собственные свойства: `enumerable`, `configurable`, `value`, `writable`, `get` и `set`. Если возвращаемый объект будет содержать еще какое-либо собственное свойство кроме перечисленных, это вызовет ошибку, как показано ниже:

```
let proxy = new Proxy({}, {  
  getOwnPropertyDescriptor(trapTarget, key) {  
    return {  
      name: "proxy";  
    };  
  }  
});  
  
// вызовет ошибку  
let descriptor = Object.getOwnPropertyDescriptor(proxy, "name");
```

Свойство `name` не входит в число допустимых в дескрипторе, поэтому, когда произойдет вызов `Object.getOwnPropertyDescriptor()`, ловушка `getOwnPropertyDescriptor` вернет значение, вызывающее ошибку. Это ограничение гарантирует, что значение, возвращаемое методом `Object.getOwnPropertyDescriptor()`, всегда имеет определенную структуру независимо от методов, используемых в прокси-объектах.

Дубликаты методов для операций с дескрипторами

И снова в ECMAScript 6 имеется несколько похожих методов. На первый взгляд кажется, что методы `Object.defineProperty()` и `Object.getOwnPropertyDescriptor()` делают то же самое, что и соответствующие им методы `Reflect.defineProperty()` и `Reflect.getOwnPropertyDescriptor()`. Подобно другим парам методов, которые обсуждались выше в этой главе, эти четыре метода имеют некоторые тонкие, но важные отличия.

Методы `defineProperty()`

Методы `Object.defineProperty()` и `Reflect.defineProperty()` отличаются возвращаемыми значениями. Метод `Object.defineProperty()` возвращает свой первый аргумент, тогда как `Reflect.defineProperty()` возвращает `true` в случае успеха операции и `false` — в противном случае. Например:

```
let target = {};
```

```
let result1 = Object.defineProperty(target, "name", { value: "target" });

console.log(target === result1);    // true

let result2 = Reflect.defineProperty(target, "name", { value: "reflect" });

console.log(result2);               // true
```

Когда вызывается метод `Object.defineProperty()`, он возвращает свой аргумент `target`. Когда вызывается метод `Reflect.defineProperty()`, он возвращает значение `true`, указывающее, что операция выполнена успешно. Так как ловушка `defineProperty` должна возвращать логическое значение, для реализации поведения по умолчанию лучше всего использовать вызов `Reflect.defineProperty()`.

Методы `getOwnPropertyDescriptor()`

Метод `Object.getOwnPropertyDescriptor()` преобразует свой первый аргумент в объект, если он является простым значением, и затем продолжает работу. Метод `Reflect.getOwnPropertyDescriptor()`, напротив, возбudit ошибку, если в первом аргументе получит простое значение. Следующие пример демонстрирует обе ситуации:

```
let descriptor1 = Object.getOwnPropertyDescriptor(2, "name");
console.log(descriptor1);    // undefined
```

```
// вызовет ошибку
let descriptor = Reflect.getOwnPropertyDescriptor(2, "name");
```

Метод `Object.getOwnPropertyDescriptor()` вернул `undefined`, потому что преобразовал число 2 в объект, который не имеет свойства `name`. Это стандартное поведение метода, когда свойство с указанным именем отсутствует в объекте. Однако вызов `Reflect.getOwnPropertyDescriptor()` привел сразу же к ошибке, потому что этот метод не допускает передачу простых значений в первом аргументе.

Ловушка `ownKeys`

Ловушка `ownKeys` перехватывает вызов внутреннего метода `[[OwnPropertyKeys]]` и позволяет переопределять это поведение, возвращая массив значений. Этот массив используется четырьмя методами: `Object.keys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()` и `Object.assign()`. (Метод `Object.assign()` использует массив, чтобы определить, какие свойства копировать.)

Поведение по умолчанию ловушки `ownKeys` реализует метод `Reflect.ownKeys()`, возвращающий массив со всеми ключами собственных свойств, включая строки и символы. Методы `Object.getOwnPropertyNames()` и `Object.keys()` исключают символьные ключи из массива и возвращают получившийся результат, а метод `Object.getOwnPropertySymbols()`, наоборот, исключает из массива строковые ключи. Метод `Object.assign()` использует все имеющиеся в массиве ключи — и строковые, и символьные.

Ловушка `ownKeys` принимает единственный аргумент `target` и должна вернуть объект, подобный массиву; в противном случае будет возбуждена ошибка. Ловушку `ownKeys` можно использовать, например, для фильтрации свойств с определенными ключами, которые не должны обнаруживаться методами `Object.keys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()` и `Object.assign()`. Представьте, что вам требуется исключить любые свойства, имена которых начинаются с символа подчеркивания — типичный способ обозначения приватных свойств в JavaScript. Для этого можно было бы использовать ловушку `ownKeys`, как показано ниже:

```
let proxy = new Proxy({}, {
  ownKeys(trapTarget) {
    return Reflect.ownKeys(trapTarget).filter(key => {
      return typeof key !== "string" || key[0] !== "_";
    });
  }
});
let nameSymbol = Symbol("name");

proxy.name = "proxy";
proxy._name = "private";
proxy[nameSymbol] = "symbol";

let names = Object.getOwnPropertyNames(proxy),
    keys = Object.keys(proxy),
    symbols = Object.getOwnPropertySymbols(proxy);

console.log(names.length);      // 1
console.log(names[0]);          // "proxy"

console.log(keys.length);       // 1
console.log(keys[0]);           // "proxy"

console.log(symbols.length);    // 1
console.log(symbols[0]);        // "Symbol(name)"
```

Этот пример использует ловушку `ownKeys`, которая сначала вызывает `Reflect.ownKeys()`, чтобы получить полный список ключей для целевого объекта. Затем вызывается метод `filter()`, чтобы отфильтровать ключи, которые являются строками и начинаются с символа подчеркивания. Затем в прокси-объект добавляются три свойства: `name`, `_name` и `nameSymbol`. Когда к прокси-объекту применяются методы `Object.getOwnPropertyNames()` и `Object.keys()`, они возвращают только свойство `name`. Аналогично, когда применяется метод `Object.getOwnPropertySymbols()`, он возвращает только свойство `nameSymbol`. Свойство `_name` не появляется ни в одном из результатов, потому что оно отфильтровывается ловушкой.

Хотя ловушка `ownKeys` позволяет изменить набор ключей, возвращаемый некоторыми операциями, она не влияет на результат некоторых других операций, часто используемых в программах, таких как цикл `for-of` и метод `Object.keys()`. Эти операции нельзя перехватить в прокси-объектах. Ловушка `ownKeys` также воздействует на цикл `for-in`, который вызывает ловушку, чтобы определить, какие ключи использовать в цикле.

Обработка вызовов функций с помощью ловушек `apply` и `construct`

Из всех ловушек только `apply` и `construct` требуют, чтобы целевой объект был функцией. Как рассказывалось в главе 3, функции имеют два внутренних метода — `[[Call]]` и `[[Construct]]`, — которые выполняются при вызове функции без оператора `new` или с ним соответственно. Ловушки `apply` и `construct` соответствуют этим внутренним методам и позволяют переопределять их. Когда функция вызывается без `new`, ловушка `apply` принимает (так же, как метод `Reflect.apply()`) следующие аргументы:

- `trapTarget`. Вызываемая функция (цель для прокси-объекта).

- `thisArg`. Значение ссылки `this` внутри функции в течение вызова.
- `argumentsList`. Массив аргументов, переданных в функцию.

Ловушка `construct`, которая выполняется, когда вызов функции производится с оператором `new`, принимает следующие аргументы:

- `trapTarget`. Вызываемая функция (цель для прокси-объекта),
- `argumentsList`. Массив аргументов, переданных в функцию.

Метод `Reflect.construct()` тоже принимает эти два аргумента и третий необязательный аргумент `newTarget`. Когда аргумент `newTarget` присутствует, он определяет значение `new.target` внутри функции.

Ловушки `apply` и `construct` вместе полностью контролируют поведение любой вызываемой функции. Ниже показано, как симитировать действие функции по умолчанию:

```
let target = function() { return 42 },
    proxy = new Proxy(target, {
      apply: function(trapTarget, thisArg, argumentList) {
        return Reflect.apply(trapTarget, thisArg, argumentList);
      },
      construct: function(trapTarget, argumentList) {
        return Reflect.construct(trapTarget, argumentList);
      }
    });

// прокси-объект, целью которого является функция, выглядит как функция
console.log(typeof proxy);           // "function"

console.log(proxy());                // 42

var instance = new proxy();
console.log(instance instanceof proxy); // true
console.log(instance instanceof target); // true
```

Здесь у нас имеется функция, возвращающая число 42. Прокси-объект для этой функции определяет ловушки `apply` и `construct`, которые делегируют поведение по умолчанию методам `Reflect.apply()` и `Reflect.construct()` соответственно. В результате функция `proxy` действует в точности как целевая функция `target`, и даже оператор `typeof` идентифицирует ее как функцию. Сначала функция `proxy` вызывается без ключевого слова `new`, чтобы получить 42, а затем с ключевым словом `new`, чтобы создать объект с именем `instance`. Этот объект интерпретируется как экземпляр обеих функций, `proxy` и `target`, потому что для определения принадлежности `instanceof` использует цепочку прототипов. Поиск в цепочке прототипов не перехватывается в этом прокси-объекте, поэтому `proxy` и `target` выглядят как имеющие один и тот же прототип.

Проверка параметров функции

Ловушки `apply` и `construct` открывают дополнительные возможности, позволяющие влиять на ход выполнения функции. Например, допустим, что требуется гарантия того, что все аргументы функции имеют определенный тип. Такую проверку аргументов можно выполнить с помощью ловушки `apply`:

```
// складывает все аргументы
function sum(...values) {
    return values.reduce((previous, current) => previous + current, 0);
}

let sumProxy = new Proxy(sum, {
    apply: function(trapTarget, thisArg, argumentList) {

        argumentList.forEach((arg) => {
            if (typeof arg !== "number") {
                throw new TypeError("All arguments must be numbers.");
            }
        });

        return Reflect.apply(trapTarget, thisArg, argumentList);
    },
    construct: function(trapTarget, argumentList) {
        throw new TypeError("This function can't be called with new.");
    }
});

console.log(sumProxy(1, 2, 3, 4));    // 10

// вызовет ошибку
console.log(sumProxy(1, "2", 3, 4));

// также вызовет ошибку
let result = new sumProxy();
```

Этот пример демонстрирует, как с помощью ловушки `apply` гарантировать, что все аргументы функции являются числами. Функция `sum()` складывает все полученные аргументы. Если ей передать нечисловое значение, она все равно попытается выполнить операцию, но результат может получиться неожиданным. Прокси-объект `sumProxy`, в который заключена функция `sum()`, перехватывает вызов функции и проверяет каждый аргумент. Если все аргументы являются числами, вызывается оригинальная функция. Для большей надежности в прокси-объекте определена также ловушка `construct`, гарантирующая невозможность вызова с ключевым словом `new`.

Аналогично можно гарантировать вызов только с ключевым словом `new` и проверить, что все аргументы являются числами:

```
function Numbers(...values) {
    this.values = values;
}

let NumbersProxy = new Proxy(Numbers, {
    apply: function(trapTarget, thisArg, argumentList) {
        throw new TypeError("This function must be called with new.");
    },
    construct: function(trapTarget, argumentList) {
        argumentList.forEach((arg) => {
            if (typeof arg !== "number") {
                throw new TypeError("All arguments must be numbers.");
            }
        });
    }
});
```



```
        return Reflect.construct(trapTarget, argumentList);
    }
});

let instance = new NumbersProxy(1, 2, 3, 4);
console.log(instance.values);    // [1,2,3,4]

// вызовет ошибку
NumbersProxy(1, 2, 3, 4);
```

Здесь ловушка `apply` возбуждает ошибку, а ловушка `construct` выполняет проверку аргументов и вызывает метод `Reflect.construct()`, чтобы вернуть новый экземпляр. Конечно, все то же самое можно реализовать без применения прокси-объекта с помощью `new.target`.

Вызов конструкторов без ключевого слова `new`

В главе 3 было представлено метасвойство `new.target`. Просто напомним, что `new.target` — это ссылка на функцию, вызванную с ключевым словом `new`. Значит, проверив значение `new.target`, можно узнать, была ли вызвана функция с ключевым словом `new` или без него, например:

```
function Numbers(...values) {

    if (typeof new.target === "undefined") {
        throw new TypeError("This function must be called with new.");
    }

    this.values = values;
}

let instance = new Numbers(1, 2, 3, 4);
console.log(instance.values);    // [1,2,3,4]

// вызовет ошибку
Numbers(1, 2, 3, 4);
```

Этот код возбудит ошибку при попытке вызвать `Numbers()` без ключевого слова `new`. Своим действием этот пример напоминает второй пример из раздела «Проверка параметров функции» выше, но не использует прокси-объект. Такой код писать намного проще, чем код, использующий прокси-объект, и данный подход предпочтительнее, если единственной целью является предотвращение возможности вызова функции без ключевого слова `new`. Но иногда функции, чье поведение требуется модифицировать, пишут другие. В таких случаях применение прокси-объекта вполне оправданно.

Представьте, что функция `Numbers()` определена в коде, который вы не можете изменить. Вам известно, что код опирается на метасвойство `new.target`, и требуется обойти эту проверку, чтобы обеспечить успешное выполнение функции. Поведение функции с ключевым словом `new` уже определено, поэтому вам остается только реализовать ловушку `apply`:

```
function Numbers(...values) {

    if (typeof new.target === "undefined") {
        throw new TypeError("This function must be called with new.");
    }

    this.values = values;
}
```

```
    this.values = values;
  }

let NumbersProxy = new Proxy(Numbers, {
  apply: function(trapTarget, thisArg, argumentsList) {
    return Reflect.construct(trapTarget, argumentsList);
  }
});

let instance = NumbersProxy(1, 2, 3, 4);
console.log(instance.values);    // [1,2,3,4]
```

Функция `NumbersProxy()` позволяет вызвать `Numbers()` без ключевого слова `new` и заставить ее действовать так, как если бы она была вызвана с `new`. Для этого ловушка `apply` вызывает `Reflect.construct()` с аргументами, переданными в `apply`. Метасвойство `new.target` внутри `Numbers()` будет хранить ссылку на `Numbers()`, поэтому ошибки не возникнет. Это был простой пример изменения `new.target`, но то же самое можно сделать еще более прямолинейным способом.

Переопределение конструкторов абстрактных базовых классов

Можно сделать еще шаг вперед и передать в вызов `Reflect.construct()` третий аргумент со значением для `new.target`. Этот прием может пригодиться, когда функция сравнивает `new.target` с известным значением, например, в конструкторе абстрактного базового класса (обсуждается в главе 9). Как ожидается, в конструкторе абстрактного базового класса метасвойство `new.target` будет иметь значение, отличное от значения в конструкторе этого класса, например:

```
class AbstractNumbers {
  constructor(...values) {
    if (new.target === AbstractNumbers) {
      throw new TypeError("This function must be inherited from.");
    }

    this.values = values;
  }
}

class Numbers extends AbstractNumbers {}

let instance = new NumbersQ, 2, 3, 4);
console.log(instance.values);    // [1,2,3,4]

// вызовет ошибку
new AbstractNumbers(1, 2, 3, 4);
```

В вызове `new AbstractNumbers()` метасвойство `new.target` будет хранить ссылку на `AbstractNumbers`, в результате чего будет возбуждена ошибка. Однако вызов `new Numbers()` выполнится успешно, потому что `new.target` ссылается на `Numbers`. Это ограничение можно обойти, вручную присвоив новое значение метасвойству `new.target` в прокси-объекте:

```
class AbstractNumbers {
  constructor(...values) {
```

```
    if (new.target === AbstractNumbers) {
      throw new TypeError("This function must be inherited from.");
    }
    this.values = values;
  }
}

let AbstractNumbersProxy = new Proxy(AbstractNumbers, {
  construct: function(trapTarget, argumentList) {
    return Reflect.construct(trapTarget, argumentList, function() {});
  }
});

let instance = new AbstractNumbersProxy(1, 2, 3, 4);
console.log(instance.values); // [1,2,3,4]
```

В прокси-объекте `AbstractNumbersProxy` определена ловушка `construct`, которая перехватывает вызов метода `new AbstractNumbersProxy()`. Затем эта ловушка вызывает метод `Reflect.construct()`, передавая аргументы, полученные ловушкой, и добавляя третий аргумент со ссылкой на пустую функцию. Эта ссылка используется как значение `new.target` внутри конструктора. Так как теперь `new.target` имеет значение, отличное от `AbstractNumbers`, ошибка не возбуждается и конструктор успешно выполняется.

Вызываемые конструкторы классов

В главе 9 говорилось, что конструкторы классов всегда должны вызываться с ключевым словом `new`, потому что внутренний метод `[[Call]]` реализован так, что возбуждает ошибку. Но прокси-объекты способны перехватывать вызов метода `[[Call]]`, благодаря чему можно создавать вызываемые классы конструкторов. Например, если требуется, чтобы конструктор класса выполнялся без ошибок при вызове без ключевого слова `new`, можно в ловушке `apply` создать новый экземпляр. Следующий код демонстрирует, как это сделать:

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

let PersonProxy = new Proxy(Person, {
  apply: function(trapTarget, thisArg, argumentList) {
    return new trapTarget(...argumentList);
  }
});

let me = PersonProxy("Nicholas");
console.log(me.name); // "Nicholas"
console.log(me instanceof Person); // true
console.log(me instanceof PersonProxy); // true
```

Объект `PersonProxy` — это прокси-объект для конструктора класса `Person`. Конструкторы классов — это обычные функции, поэтому внутри прокси-объектов их можно использовать как функции. Ловушка `apply` переопределяет поведение по умолчанию и возвращает новый экземпляр `trapTarget` класса `Person`. (Я использовал `trapTarget` в этом примере, чтобы показать,

что нет необходимости вручную указывать класс.) Список аргументов `argumentList` передается в `trapTarget` с использованием оператора расширения, чтобы передать каждый аргумент по отдельности. Вызов `PersonProxy()` без ключевого слова `new` возвращает экземпляр `Person`; если попытаться вызвать `Person()` без `new`, конструктор по-прежнему возбудит ошибку. Создание вызываемых конструкторов классов — лишь один из вариантов использования прокси-объектов.

Отключение прокси-объектов

Обычно прокси-объект нельзя отключить от целевого объекта после создания прокси-объекта. Все примеры, приводившиеся до этого момента в данной главе, использовали неотключаемые прокси-объекты. Но иногда бывает желательно отключить прокси, чтобы он больше не использовался. Возможность отключения прокси-объекта может пригодиться, когда из соображений безопасности требуется предоставить объект через API и иметь возможность выключать некоторые функциональные возможности в любой момент.

Для этого можно создавать отключаемые прокси-объекты с помощью метода `Proxy.revocable()`, который принимает те же аргументы, что и конструктор `Proxy`, а именно целевой объект и обработчик. Возвращаемое значение является объектом со следующими свойствами:

- `proxy`. Прокси-объект, который можно отключить.
- `revoke`. Функция, которая вызывается для отключения прокси-объекта.

После вызова функции `revoke()` прокси-объект прекращает перехватывать операции. Любые попытки использовать прокси-объект таким способом, чтобы вызвать его ловушки, приводят к ошибке. Например:

```
let target = {
  name: "target"
};

let { proxy, revoke } = Proxy.revocable(target, {});

console.log(proxy.name);    // "target"

revoke();

// вызовет ошибку
console.log(proxy.name);
```

Этот код создает отключаемый прокси-объект. В нем используется прием деструктуризации, чтобы присвоить переменным `proxy` и `revoke` значения одноименных свойств объекта, возвращаемого методом `Proxy.revocable()`. После этого объект `proxy` можно использовать как неотключаемый прокси-объект, поэтому обращение к свойству `proxy.name` возвращает `"target"`, которое фактически преобразуется в обращение к свойству `target.name`. Однако после вызова функции `revoke()` объект `proxy` прекращает функционировать. Попытка получить значение `proxy.name` вызывает ошибку, равно как и любые другие попытки выполнить операции, вызывающие ловушки в объекте `proxy`.

Решение проблемы с массивами

В начале этой главы я рассказал, что до появления ECMAScript 6 разработчики не имели возможности в точности воспроизвести поведение массивов. С помощью прокси-объектов и `Reflection`

API можно создать объект, который при добавлении и удалении его свойств будет действовать точно так же, как встроенный тип `Array`. Чтобы освежить память, следующий пример демонстрирует поведение, которое помогают симитировать прокси-объекты:

```
let colors = ["red", "green", "blue"];

console.log(colors.length);    // 3
colors[3] = "black";

console.log(colors.length);    // 4
console.log(colors[3]);        // "black"

colors.length = 2;
console.log(colors.length);    // 2
console.log(colors[3]);        // undefined
console.log(colors[2]);        // undefined
console.log(colors[1]);        // "green"
```

Обратите внимание на два наиболее важных аспекта в этом примере:

- Свойство `length` увеличивается до 4 после присваивания значения элементу `colors[3]`.
- Последние два элемента удаляются из массива, когда свойству `length` присваивается значение 2.

Именно эти два аспекта, и только их, требуется симитировать, чтобы точно воссоздать поведение встроенных массивов. Следующие несколько разделов описывают, как создать объект, который точно имитирует их.

Определение индексов массива

Имейте в виду, что присваивание значения свойству с целочисленным ключом является особым случаем для массивов, потому что они интерпретируются иначе, чем нецелочисленные ключи. Спецификация ECMAScript 6 содержит следующие инструкции о том, как определить, является ли ключ свойства индексом массива:

Строковое имя `P` свойства является индексом массива тогда и только тогда, если `ToString(ToUint32(P))` равно `P` и `ToUint32(P)` ниже $2^{32} - 1$.

Далее показано, как эту проверку реализовать на языке JavaScript:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}
```

Функция `toUint32()` преобразует заданное значение в 32-разрядное целое без знака с применением алгоритма, описанного в спецификации. Функция `isArrayIndex()` сначала преобразует ключ в значение `uint32`, а затем выполняет сравнение, чтобы определить, является ли ключ индексом массива. Имея эти две вспомогательные функции, можно приступить к реализации объекта, имитирующего встроенный массив.

Увеличение значения `length` при добавлении новых элементов

Обратите внимание, что оба аспекта поведения массивов, описанные выше, связаны с операцией присваивания значений свойствам. Значит, для реализации обоих аспектов нам понадобится определить только ловушку `set`. Для начала рассмотрим пример, реализующий первый из двух аспектов — увеличение свойства `length` при использовании индекса массива, который больше, чем `length - 1`:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

function createMyArray(length=0) {
    return new Proxy({ length }, {
        set(trapTarget, key, value) {

            let currentLength = Reflect.get(trapTarget, "length");

            // специальный случай
            if (isArrayIndex(key)) {
                let numericKey = Number(key);

                if (numericKey >= currentLength) {
                    Reflect.set(trapTarget, "length", numericKey + 1);
                }
            }

            // следующая инструкция выполняется всегда
            // независимо от типа ключа
            return Reflect.set(trapTarget, key, value);
        }
    });
}

let colors = createMyArray(3);
console.log(colors.length);    // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
console.log(colors.length);    // 3

colors[3] = "black";

console.log(colors.length);    // 4
console.log(colors[3]);        // "black"
```

Здесь используется ловушка `set` прокси-объекта, чтобы перехватить операцию присваивания по индексу массива. Если ключ является индексом массива, он преобразуется в число, потому

что ключи всегда передаются в виде строк. Далее, если это числовое значение больше или равно текущему значению свойства `length`, свойству `length` присваивается число на единицу больше числового ключа (присваивание элементу в позиции 3 предполагает, что свойство `length` должно получить значение 4). Затем вызовом `Reflect.set()` выполняется операция присваивания по умолчанию, потому что нам необходимо, чтобы свойство получило указанное значение.

Первоначально нестандартный массив создается вызовом `createMyArray()` со значением `length`, равным 3, после чего сразу же инициализируются три первых элемента. Свойство `length` сохраняет свое значение 3, пока не будет присвоено значение `"black"` элементу в позиции 3. В этот момент свойству `length` присваивается значение 4.

После реализации первого аспекта поведения можно приступить ко второму.

Удаление элементов при уменьшении значения `length`

Первый аспект поведения массивов используется, только когда индекс больше или равен значению свойства `length`. Второй аспект действует с точностью до наоборот: когда свойству `length` присваивается значение, которое меньше текущего, это приводит к удалению элементов массива. Данный аспект вовлекает не только изменение свойства `length`, но также удаление всех элементов, которые могут существовать. Например, если свойству `length` массива с 4 элементами присвоить значение 2, это должно повлечь удаление элементов в позициях 2 и 3. Этот аспект также можно реализовать в ловушке `set` прокси-объекта наряду с первым аспектом. Ниже вновь приводится первый пример, но с измененным методом `createMyArray`:

```
function toUint32(value) {
    return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
    let numericKey = toUint32(key);
    return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

function createMyArray(length=0) {
    return new Proxy( length , {
        set(trapTarget, key, value) {

            let currentLength = Reflect.get(trapTarget, "length");

            // специальный случай
            if (isArrayIndex(key)) {
                let numericKey = Number(key);

                if (numericKey >= currentLength) {
                    Reflect.set(trapTarget, "length", numericKey + 1);
                }
            } else if (key === "length") {

                if (value < currentLength) {
                    for (let index = currentLength - 1; index >= value;
                        index--) {
                        Reflect.deleteProperty(trapTarget, index);
                    }
                }
            }
        }
    });
}
```

```
    }  
  }  
  
  // следующая инструкция выполняется всегда  
  // независимо от типа ключа  
  return Reflect.set(trapTarget, key, value);  
}  
});  
}  
  
let colors = createMyArray(3);  
console.log(colors.length);    // 3  
  
colors[0] = "red";  
colors[1] = "green";  
colors[2] = "blue";  
colors[3] = "black";  
  
console.log(colors.length);    // 4  
  
colors.length = 2;  
  
console.log(colors.length);    // 2  
console.log(colors[3]);        // undefined  
console.log(colors[2]);        // undefined  
console.log(colors[1]);        // "green"  
console.log(colors[0]);        // "red"
```

Ловушка `set` в этом примере сравнивает ключ со строкой `"length"` и, если условие выполняется, корректирует содержимое объекта. Сначала вызовом `Reflect.get()` извлекается текущая длина массива и сравнивается с новым значением. Если новое значение меньше текущей длины, цикл `for` удаляет все свойства в целевом объекте, которые должны быть недоступны. Цикл `for` выполняет обход в обратном направлении от текущей длины массива (`currentLength`) и удаляет каждое свойство, пока не будет достигнута новая длина массива (`value`).

В этом примере в `colors` добавляются четыре цвета, затем свойству `length` присваивается значение 2. Это приводит к удалению элементов в позициях 2 и 3, то есть теперь любые попытки обращения к ним будут возвращать `undefined`. Свойство `length` получает значение 2, и элементы в позициях 0 и 1 остаются доступными.

Реализовав оба аспекта поведения, вы теперь легко сможете создать объект, имитирующий поведение встроенных массивов. Но реализация в виде функции выглядит не особенно привлекательно; гораздо удобнее иметь класс, инкапсулирующий это поведение, поэтому следующим шагом реализуем такой класс.

Реализация класса `MyArray`

Простейший способ создать класс, использующий прокси-объект, заключается в том, чтобы определить обычный класс, конструктор которого возвращает прокси-объект. Таким образом, при попытке создания экземпляра будет возвращаться прокси-объект, а не сам экземпляр. (Экземпляр — это значение `this` внутри конструктора.) Экземпляр в этом случае станет целевым объектом для прокси, а сам прокси-объект будет возвращен, как если бы он был экземпляром. Экземпляр будет полностью скрыт от приложения, и вы не сможете обратиться к нему непосредственно, но сможете пользоваться им косвенно через прокси-объект. Ниже приводится простой пример возврата прокси-объекта из конструктора:


```
class Thing {
  constructor() {
    return new Proxy(this, {});
  }
}

let myThing = new Thing();
console.log(myThing instanceof Thing); // true
```

В этом примере конструктор класса `Thing` возвращает прокси-объект. В качестве целевого объекта прокси-объекту передается `this`, и конструктор возвращает прокси-объект. Это означает, что `myThing` в действительности ссылается на прокси-объект, даже если он создан вызовом конструктора класса `Thing`. Так как прокси-объекты делегируют выполнение операций своим целевым объектам, `myThing` можно считать экземпляром `Thing`, а прокси-объект оказывается абсолютно прозрачным для любого кода, использующего класс `Thing`.

Учитывая, что конструктор может вернуть прокси-объект, мы относительно легко можем реализовать свою версию класса массивов с применением прокси. Реализация будет выглядеть практически так же, как в разделе «Удаление элементов при уменьшении значения `length`», приведенном выше. Реализация прокси-объекта останется прежней, но на этот раз она должна находиться внутри конструктора класса. Ниже приводится законченный пример:

```
function toUint32(value) {
  return Math.floor(Math.abs(Number(value))) % Math.pow(2, 32);
}

function isArrayIndex(key) {
  let numericKey = toUint32(key);
  return String(numericKey) == key && numericKey < (Math.pow(2, 32) - 1);
}

class MyArray {
  constructor(length=0) {
    this.length = length;

    return new Proxy(this, {
      set(trapTarget, key, value) {

        let currentLength = Reflect.get(trapTarget, "length");

        // специальный случай
        if (isArrayIndex(key)) {
          let numericKey = Number(key);

          if (numericKey >= currentLength) {
            Reflect.set(trapTarget, "length", numericKey + 1);
          }
        } else if (key === "length") {

          if (value < currentLength) {
            for (let index = currentLength - 1; index >= value;
                index--) {
              Reflect.deleteProperty(trapTarget, index);
            }
          }
        }
      }
    });
  }
}
```

```

        }
    }

    // следующая инструкция выполняется всегда
    // независимо от типа ключа
    return Reflect.set(trapTarget, key, value);
}

});
}

let colors = new MyArray(3);
console.log(colors instanceof MyArray);    // true

console.log(colors.length);                // 3

colors[0] = "red";
colors[1] = "green";
colors[2] = "blue";
colors[3] = "black";
console.log(colors.length);                // 4

colors.length = 2;

console.log(colors.length);                // 2
console.log(colors[3]);                    // undefined
console.log(colors[2]);                    // undefined
console.log(colors[1]);                    // "green"
console.log(colors[0]);                    // "red"

```

Этот код реализует класс `MyArray`, конструктор которого возвращает прокси-объект. Конструктор создает свойство `length` (и инициализирует его значением аргумента, переданного конструктору, или значением по умолчанию, равным 0) и затем создает и возвращает прокси-объект. В результате переменная `colors` выглядит как экземпляр `MyArray` и реализует оба аспекта поведения ключей массива.

Вернуть прокси-объект из конструктора класса не составляет труда, но это также означает, что для каждого экземпляра класса будет создаваться свой прокси-объект. Однако существует возможность использовать один прокси-объект для всех экземпляров. Для этого достаточно лишь использовать прокси-объект в качестве прототипа.

Использование прокси-объекта в качестве прототипа

Прокси-объект можно использовать в качестве прототипа, но такая реализация будет выглядеть сложнее, чем предыдущие примеры в этой главе. Когда прокси-объект играет роль прототипа, его ловушки вызываются, только когда операции по умолчанию выполняются прототипом, что ограничивает возможности прокси как прототипа. Рассмотрим следующий пример.

```

let target = {};
let newTarget = Object.create(new Proxy(target, {

    // этот вызов никогда не будет выполнен
    defineProperty(trapTarget, name, descriptor) {

```

```
        // возбуждит ошибку, если будет вызван
        return false;
    }
    });
Object.defineProperty(newTarget, "name", {
    value: "newTarget"
});

console.log(newTarget.name);           // "newTarget"
console.log(newTarget.hasOwnProperty("name")); // true
```

Объект `newTarget` создается с прокси-объектом в качестве прототипа. Передача `target` прокси-объекту в качестве целевого объекта фактически превращает `target` в прототип для `newTarget`, потому что прокси-объект действует прозрачно. Теперь ловушки прокси-объекта будут вызываться, только если выполнение операции с `newTarget` будет делегировано объекту `target`.

Вызов метода `Object.defineProperty()` создает в `newTarget` собственное свойство с именем `name`. Определение свойства объекта не является операцией, которая при нормальных условиях делегируется прототипу объекта, поэтому ловушка `defineProperty` в прокси-объекте никогда не будет вызвана, и свойство `name` будет добавлено в `newTarget` как собственное свойство.

Несмотря на ограничения, с которыми сталкиваются прокси-объекты, когда используются в качестве прототипов, некоторые их ловушки все же оказываются полезными. Я расскажу о них в следующих нескольких разделах.

Использование ловушки `get` в прототипе

Когда вызывается внутренний метод `[[Get]]`, чтобы прочесть свойство, он сначала пытается отыскать собственное свойство объекта. Если собственного свойства с указанным именем не обнаружится, выполняется попытка найти свойство в прототипе. Процесс продолжается, пока не останется прототипов для проверки.

Вследствие этого, если определить ловушку `get` в прокси-объекте, она будет вызываться всякий раз, когда попытка найти собственное свойство объекта завершится неудачей. Ловушку `get` можно использовать для предотвращения неожиданного поведения при обращении к свойствам, существование которых не гарантируется. Создадим объект, возбуждающий ошибку при попытке обратиться к несуществующему свойству:

```
let target = {};
let thing = Object.create(new Proxy(target, {
    get(trapTarget, key, receiver) {
        throw new ReferenceError(`${key} doesn't exist`);
    }
}));

thing, name = "thing";

console.log(thing.name); // "thing"

// вызовет ошибку
let unknown = thing.unknown;
```

В этом примере создается объект `thing`, который получает прокси-объект в качестве прототипа. Ловушка `get` возбуждает ошибку, чтобы показать, что данный ключ отсутствует в объекте `thing`.

Операция чтения свойства `thing.name` не вызывает ловушку `get` в прототипе, потому что это свойство присутствует в объекте `thing`.

Ловушка `get` вызывается, только когда происходит обращение к несуществующему свойству `thing.unknown`.

Когда выполняется последняя строка, в объекте `thing` отсутствует собственное свойство `unknown`, поэтому операция продолжается в прототипе, и ловушка `get` возбуждает ошибку. Такое поведение может очень пригодиться в языке JavaScript, который просто возвращает значение `undefined` для отсутствующих свойств вместо того, чтобы возбудить ошибку (как это происходит в других языках).

Важно понимать, что в этом примере аргументы `trapTarget` и `receiver` представляют разные объекты. Когда прокси-объект используется в качестве прототипа, в аргументе `trapTarget` передается прототип объекта, а в аргументе `receiver` — экземпляр объекта. В данном случае это означает, что `trapTarget` — это `target`, а `receiver` — это `thing`. Это позволяет обращаться к оригинальному целевому объекту и к объекту, в котором выполняется операция.

Использование ловушки `set` в прототипе

Внутренний метод `[[Set]]` также сначала пытается найти собственное свойство объекта, а затем продолжает поиск в прототипе, если это необходимо. Операция присваивания записывает значение в собственное свойство объекта с указанным именем, если оно существует. Если такого собственного свойства нет, управление передается прототипу. Вся хитрость в том, что даже когда операция продолжается в прототипе, отсутствующее свойство по умолчанию создается в экземпляре (не в прототипе) независимо от наличия в прототипе свойства с таким же именем.

Чтобы получить более полное представление о том, когда вызывается ловушка `set` в прототипе, а когда не вызывается, рассмотрим следующий пример, демонстрирующий поведение по умолчанию:

```
let target = {};
let thing = Object.create(new Proxy(target, {
  set(trapTarget, key, value, receiver) {
    return Reflect.set(trapTarget, key, value, receiver);
  }
}));

console.log(thing.hasOwnProperty("name")); // false

// приведет к вызову ловушки 'set'
thing.name = "thing";

console.log(thing.name); // "thing"
console.log(thing.hasOwnProperty("name")); // true

// ловушка `set` не будет вызвана
thing.name = "boo";

console.log(thing.name); // "boo"
```

В этом примере объект `target` первоначально не имеет собственных свойств. Объект `thing` получает в качестве прототипа прокси-объект, определяющий ловушку `set`, которая перехватывает операции создания любых новых свойств. Когда свойству `thing.name` присваивается строка `"thing"`, вызывается ловушка `set`, потому что `thing` не имеет собственного свойства с именем `name`. Внутри ловушки `set` аргумент `trapTarget` ссылается на `target`, а `receiver` — на `thing`. В конечном итоге операция должна создать новое свойство в объекте `thing`, и, к счастью, метод `Reflect.set()` реализует нужное нам поведение по умолчанию, если ему передать в четвертом аргументе объект `receiver`.

После создания свойства `name` в объекте `thing` повторная попытка изменить значение свойства `thing.name` не приводит к вызову ловушки `set`. В этот момент собственное свойство `name` уже существует, поэтому операция `[[Set]]` не продолжается в прототипе.

Использование ловушки `has` в прототипе

Как вы наверняка помните, ловушка `has` перехватывает применение оператора `in` к объектам. Оператор `in` пытается отыскать в объекте первое собственное свойство с заданным именем. Если такое свойство отсутствует, операция продолжается в прототипе. Если прототип тоже не имеет собственного свойства с таким именем, поиск продолжается в цепочке прототипов, пока не будет найдено искомое собственное свойство или не будет достигнут последний прототип в цепочке.

Следовательно, ловушка `has` вызывается, только когда операция поиска достигает прокси-объекта в цепочке прототипов. Когда прокси-объект используется в роли прототипа, ловушка `has` вызывается, только если объект не имеет собственного свойства с требуемым именем. Например:

```
let target = {};  
let thing = Object.create(new Proxy(target, {  
  has(trapTarget, key) {  
    return Reflect.has(trapTarget, key);  
  }  
}));
```

```
// приведет к вызову ловушки `has`  
console.log("name" in thing);    // false
```

```
thing.name = "thing";
```

```
// ловушка `has` не будет вызвана  
console.log("name" in thing);    // true
```

В этом примере определяется прокси-объект с ловушкой `has`, который играет роль прототипа объекта `thing`. Ловушка `has` не получает аргумент `receiver` с объектом, как ловушки `get` и `set`, потому что при использовании оператора `in` поиск в прототипе происходит автоматически. Вместо этого ловушка `has` должна оперировать только объектом `trapTarget` — целевым объектом прокси-объекта. Первая попытка использовать оператор `in` в этом примере приводит к вызову ловушки `has`, потому что в объекте `thing` отсутствует собственное свойство `name`. После присваивания значения свойству `thing.name` повторная попытка использовать оператор `in` не приводит к вызову ловушки `has`, потому что операция останавливается после обнаружения собственного свойства `name` в `thing`.

Все примеры с прототипами, представленные до сих пор, были основаны на создании объектов с помощью метода `Object.create()`. Но если потребуется создать класс, использующий прокси-объект в качестве прототипа, придется приложить дополнительные усилия.

Прокси-объекты как прототипы классов

В классах нельзя использовать прокси-объекты в качестве прототипов непосредственно, потому что их свойство `prototype` недоступно для записи. Однако можно применить обходной прием и создать класс с прокси-объектом в роли прототипа, используя наследование. Для начала нужно определить тип в стиле ECMAScript 5, используя функцию-конструктор. После этого можно переопределить прототип, как показано ниже:

```
function NoSuchProperty() {  
  // пустая
```

```
}

NoSuchProperty.prototype = new Proxy({}, {
  get(trapTarget, key, receiver) {
    throw new ReferenceError(`${key} doesn't exist`);
  }
});

let thing = new NoSuchProperty();

// вызовет ошибку в ловушке `get`
let result = thing.name;
```

Функция `NoSuchProperty` представляет базовый тип, который унаследует наш класс. Свойство `prototype` функций не имеет никаких ограничений, поэтому ему можно присвоить ссылку на прокси-объект. Цель ловушки `get` — возбуждать ошибку при попытке обратиться к несуществующему свойству. Объект `thing` создается как экземпляр `NoSuchProperty` и возбуждает ошибку при обращении к несуществующему свойству `name`.

Следующий шаг — создание класса, наследующего `NoSuchProperty`. Для внедрения прокси-объекта в цепочку прототипов класса можно просто использовать ключевое слово `extends`, обсуждавшееся в главе 9, например:

```
function NoSuchProperty() {
  // пустая
}

NoSuchProperty.prototype = new Proxy({}, {
  get(trapTarget, key, receiver) {
    throw new ReferenceError(`${key} doesn't exist`);
  }
});

class Square extends NoSuchProperty {
  constructor(length, width) {
    super();
    this.length = length;
    this.width = width;
  }
}

let shape = new Square(2, 6);

let area1 = shape.length * shape.width;
console.log(area1); // 12

// вызовет ошибку из-за отсутствия свойства "width"
let area2 = shape.length * shape.width;
```

В этом примере определяется класс `Square`, наследующий тип `NoSuchProperty`, благодаря чему прокси-объект оказывается включенным в цепочку прототипов класса `Square`. Затем создается объект `shape` как новый экземпляр `Square`, имеющий два собственных свойства: `length` и `width`. Операции чтения этих свойств выполняются успешно, потому что ловушка `get` не вызывается. Только когда выполняется попытка обратиться к несуществующему свойству объекта `shape`

(`shape.width` — очевидная опечатка), вызывается ловушка `get` в прокси-объекте и возбуждается ошибка, что доказывает присутствие прокси-объекта в цепочке прототипов объекта `shape`. Но может быть неочевидно, что прокси-объект не является прямым прототипом объекта `shape`. Фактически прокси-объект находится в цепочке прототипов в паре шагов от `shape`. В этом можно убедиться, немного изменив предыдущий пример:

```
function NoSuchProperty() {
  // пустая
}

// сохранить ссылку на прокси-объект,
// который будет использоваться как прототип
let proxy = new Proxy({}, {
  get(trapTarget, key, receiver) {
    throw new ReferenceError(`${key} doesn't exist`);
  }
});

NoSuchProperty.prototype = proxy;

class Square extends NoSuchProperty {
  constructor length, width) {
    super();
    this.length = length;
    this.width = width;
  }
}

let shape = new Square(2, 6);

let shapeProto = Object.getPrototypeOf(shape);

console.log(shapeProto === proxy);           // false

let secondLevelProto = Object.getPrototypeOf(shapeProto);

console.log(secondLevelProto === proxy);     // true
```

Эта версия кода сохраняет ссылку на прокси-объект в переменной `proxy`, чтобы упростить его идентификацию позднее. Прототип объекта `shape` хранится в свойстве `Shape.prototype` и не является прокси-объектом. Но прокси-объектом является прототип объекта `Shape.prototype`, унаследованный от `NoSuchProperty`.

Наследование добавляет еще один шаг в цепочке прототипов, и это важно, потому что операции, которые могут привести к вызову ловушки `get` в прокси-объекте, должны сделать этот дополнительный шаг, чтобы достичь ее. Если искомое свойство обнаружится в прототипе `Shape.prototype`, как в следующем примере, это предотвратит вызов ловушки `get`:

```
function NoSuchProperty() {
  // пустая
}

NoSuchProperty.prototype = new Proxy({}, {
  get(trapTarget, key, receiver) {
```

```
        throw new ReferenceError(`${key} doesn't exist`);
    }
});

class Square extends NoSuchProperty {
    constructor(length, width) {
        super();
        this.length = length;
        this.width = width;
    }
    getArea() {
        return this.length * this.width;
    }
}

let shape = new Square(2, 6);

let area1 = shape.length * shape.width;
console.log(area1);    // 12

let area2 = shape.getArea();
console.log(area2);    // 12

// вызовет ошибку из-за отсутствия свойства "width"
let area3 = shape.length * shape.width;
```

Здесь в классе `Square` имеется метод `getArea()`. Он автоматически добавляется в прототип `Square.prototype`, поэтому, когда происходит вызов `shape.getArea()`, поиск метода `getArea()` начинается в экземпляре `shape` и затем продолжается в его прототипе. Так как `getArea()` обнаруживается в прототипе, поиск прекращается и прокси-объект не вызывается. В данной ситуации требуется именно такое поведение, потому что генерировать ошибку при попытке вызвать метод `getArea()` было бы неправильно.

Несмотря на дополнительный код, который приходится писать, чтобы создать класс с прокси-объектом в цепочке прототипов, иногда получаемые выгоды стоят затраченных усилий.

В заключение

До появления ECMAScript 6 некоторые объекты (такие, как массивы) демонстрировали нестандартное поведение, которое невозможно было повторить в собственных объектах. Прокси-объекты изменили эту ситуацию. Они позволяют определять нестандартное поведение для некоторых низкоуровневых операций и с помощью их ловушек воспроизводить любые аспекты поведения встроенных объектов JavaScript. Эти ловушки вызываются за кулисами, когда выполняются разные операции, такие как применение оператора `in`.

В ECMAScript 6 появился также механизм рефлексии (Reflection API), дающий разработчикам возможность реализовать поведение по умолчанию для любой ловушки в прокси-объектах. Для каждой ловушки в объекте `Reflect` (еще одно новшество в ECMAScript 6) имеется соответствующий метод с тем же именем. Используя комбинацию из ловушек прокси-объекта и методов Reflection API, можно заставить некоторые операции вести себя иначе в одних ситуациях и действовать по умолчанию — в других.

Отключаемые прокси-объекты — это специальные прокси-объекты, которые можно эффективно отключать вызовом функции `revoke()`. Функция `revoke()` завершает функционирование прокси-объекта, поэтому любые попытки взаимодействий со свойствами прокси после ее вызова приводят к

ошибке. Отключаемые прокси-объекты играют важную роль в обеспечении безопасности приложений, когда сторонним разработчикам может понадобиться доступ к объектам в течение некоторого интервала времени.

Прямое использование прокси-объектов — наиболее действенный путь, но иногда бывает желательно использовать их в роли прототипов других объектов. Однако в этом случае уменьшается число ловушек, сохраняющих свою практическую ценность. Только ловушки `get`, `set` и `has` могут вызываться в ограниченном числе случаев, когда прокси-объект используется в качестве прототипа.

Инкапсуляция кода в модули

Общедоступность загружаемого кода является одним из самых удручающих и способствующих ошибкам аспектов языка JavaScript. В других языках используются такие понятия, как пакеты, помогающие ограничить область видимости кода, но в JavaScript до появления ECMAScript 6 все, что определялось в каждом файле JavaScript, автоматически становилось доступным в глобальной области видимости. С ростом сложности веб-приложений и используемого ими объема кода на JavaScript такой подход все чаще стал вызывать проблемы, такие как конфликты имен и недостаточный уровень безопасности. Одной из целей ECMAScript 6 было решение проблемы области видимости и наведение порядка в приложениях на JavaScript. Так появились модули.

Что такое модули?

Модуль — это фрагмент кода на JavaScript, который автоматически действует в строгом режиме без возможности изменить его. В противоположность архитектуре с общедоступными элементами, переменные, созданные на верхнем уровне модуля, не становятся автоматически доступными в общей, глобальной области видимости. Переменные существуют только в пределах области видимости модуля, и модуль должен экспортировать свои компоненты, такие как переменные или функции, которые должны быть доступны коду за его пределами. Модули могут также импортировать компоненты из других модулей.

Две другие особенности модулей в меньшей степени связаны с ограничением области видимости, но тем не менее имеют важное значение. Первая: ссылка `this` на верхнем уровне модуля имеет значение `undefined`. Вторая: модули не поддерживают комментарии в стиле HTML внутри кода, которые являются рудиментами, оставшимися от ранних реализаций JavaScript в браузерах.

Сценарии, подключающие любой код на JavaScript, который не является модулем, не обладают этими особенностями. Различия между модулями и другим кодом на JavaScript на первый взгляд могут показаться незначительными, но они существенно влияют на порядок загрузки и выполнения кода на JavaScript, о чем я подробно буду рассказывать в этой главе. Истинная мощь модулей проявляется в возможности экспортировать и импортировать только необходимые компоненты, а не все, что определено в файле. Хорошее знание механизмов экспортирования и импортирования совершенно необходимо для понимания отличий между модулями и сценариями.

Основы экспортирования

Доступ к компонентам модуля извне можно открыть с помощью ключевого слова `export`. В простейшем случае достаточно поместить `export` перед объявлением переменной, функции или

класса, чтобы экспортировать их из модуля, например:

```
// экспортировать данные
export var color = "red";
export let name = "Nicholas";
export const magicNumber = 7;

// экспортировать функцию
export function sum(num1, num2) {
    return num1 + num1;
}

// экспортировать класс
export class Rectangle {
    constructor(length, width) {
        this.length = length;
        this.width = width;
    }
}

// эта функция доступна только внутри модуля
function subtract(num1, num2) {
    return num1 - num2;
}

// определить функцию...
function multiply(num1, num2) {
    return num1 * num2;
}

// ...и экспортировать ее
export multiply;
```

В этом примере есть несколько моментов, на которые следует обратить внимание. Кроме ключевого слова **export** объявления ничем не отличаются от объявлений в простом сценарии. Все экспортируемые функции и классы имеют имена, потому что это необходимо для экспорта. Анонимные функции или классы с помощью этого синтаксиса можно экспортировать только с использованием ключевого слова **default** (подробно обсуждается в разделе «Значения по умолчанию в модулях» ниже).

Обратите также внимание на функцию **multiply()**, которая не экспортируется во время определения. Такой способ тоже допустим, потому что не всегда требуется экспортировать объявление: можно экспортировать ссылки. Кроме того, отметьте, что функция **subtract()** в этом примере не экспортируется. Она будет недоступна за пределами модуля, потому что любые переменные, функции и классы, не экспортированные явно, остаются закрытыми компонентами, доступными только в модуле.

Основы импортирования

При наличии модуля, экспортирующего свои компоненты, доступ к этим компонентам в другом модуле можно открыть с помощью ключевого слова **import**. Инструкция **import** состоит из двух частей: первая определяет импортируемые идентификаторы, а вторая — модуль, из которого они импортируются.

Ниже приводится базовая форма этой инструкции:

```
import { identifier1 identifier2 } from "./example.js";
```

Фигурные скобки после `import` определяют список привязок, импортируемых из заданного модуля. Ключевое слово `from` определяет модуль, из которого импортируются привязки. Модуль определяется строкой, представляющей путь к модулю (называется *спецификатором модуля*). Браузеры используют тот же формат путей, который применяется в элементах `<script>`, и поэтому требуется указывать расширение файла. В Node.js, однако, используются другие соглашения, позволяющие отличать локальные файлы и пакеты по префиксу файловой системы. Например, имя `example` интерпретировалось бы как имя пакета, а `./example.js` — как имя локального файла.

ПРИМЕЧАНИЕ

Список привязок в инструкции `import` выглядит как операция деструктуризации объекта, но это не так.

Когда вы импортируете привязки из модуля, они действуют, как если бы они были объявлены с ключевым словом `const`. В результате вы не сможете объявить другую переменную с тем же именем (а также импортировать другую привязку с тем же именем), использовать идентификатор до инструкции `import` или изменить значение привязки.

Импортирование единственной привязки

Представьте, что пример из раздела «Основы экспортирования» выше — это модуль, хранящийся в файле `example.js`. Привязки из этого модуля можно импортировать и использовать множеством способов. Например, можно импортировать только один идентификатор:

```
// импортировать единственный идентификатор
import { sum } from "./example.js";
```

```
console.log(sum(1, 2)); // 3
```

```
sum = 1; // вызовет ошибку
```

Хотя помимо этой функции `example.js` экспортирует еще несколько привязок, данный пример импортирует только функцию `sum()`. Попытка связать имя `sum` с новым значением вызовет ошибку, потому что импортированные привязки нельзя изменять.

ПРИМЕЧАНИЕ

Не забудьте добавить `/`, `./` или `../` в начало строки с именем импортируемого файла для лучшей совместимости с разными версиями браузеров и Node.js.

Импортирование нескольких привязок

Чтобы импортировать несколько привязок из модуля `example`, их следует перечислить явно, как показано ниже:

```
// импортировать несколько привязок
import { sum, multiply, magicNumber } from "./example.js";
console.log(sum(1, magicNumber)); // 8
console.log(multiply(1, 2)); // 2
```

Этот пример импортирует три привязки из модуля `example`: `sum`, `multiply` и `magicNumber`. Затем они используются, как если бы были определены локально.

Импортирование всего модуля

Особый случай позволяет импортировать модуль целиком как единый объект. В этом случае все компоненты, экспортируемые модулем, будут доступны как свойства такого объекта. Например:

```
// импортировать все
import * as example from "./example.js";
console.log(example.sum(1,
    example.magicNumber));           // 8
console.log(example.multiply(1, 2)); // 2
```

В этом примере в объект с именем `example` загружаются все привязки, экспортируемые из `example.js`. После этого все именованные экспортированные компоненты (в том числе функции `sum()` и `multiply()` и переменная `magicNumber`) будут доступны как свойства `example`.

Эта форма импортирования называется *импортированием пространства имен*, потому что объект `example` отсутствует внутри файла `example.js` и создается специально, чтобы играть роль объекта пространства имен для всех компонентов, экспортируемых из `example.js`.

Но имейте в виду, что независимо от количества инструкций `import`, в которых используется имя модуля, этот модуль будет выполнен только один раз. После того как первая инструкция импорта выполнит модуль, она сохранит его экземпляр в памяти для повторного использования в других инструкциях `import`. Взгляните на следующий пример:

```
import { sum } from "./example.js";
import { multiply } from "./example.js";
import { magicNumber } from "./example.js";
```

Несмотря на то что в данном модуле имеется три инструкции `import`, файл `example.js` выполнится только один раз. Если привязки из `example.js` попытается импортировать другой модуль в этом же приложении, он получит тот же экземпляр модуля, который использует данный код.

ОГРАНИЧЕНИЯ СИНТАКСИСА МОДУЛЕЙ

Обе инструкции, `export` и `import`, имеют важное ограничение: они должны использоваться за пределами других инструкций и функций. Например, следующий код вызовет синтаксическую ошибку:

```
if (flag) {
    export flag;    // синтаксическая ошибка
}
```

Инструкция `export` находится внутри инструкции `if`, что недопустимо. Операция экспортирования не может выполняться по условию или динамически каким-либо образом. Одна из причин внедрения поддержки модулей состояла в том, чтобы дать возможность движку JavaScript статически определять экспортируемые привязки. По сути, инструкцию `export` можно использовать только на верхнем уровне модуля.

То же относится к инструкции `import` — ее можно использовать только на верхнем уровне модуля. Значит, следующий код также вызовет синтаксическую ошибку:

```
function tryImport() {
    import flag from "./example.js";    // синтаксическая ошибка
}
```

Динамическое импортирование привязок невозможно по тем же причинам. Ключевые слова `export` и `import` реализованы статически специально, чтобы такие инструменты, как текстовые редакторы, легко могли определить, какая информация из модуля будет доступна.

Тонкая особенность импортированных привязок

Инструкция `import` в ECMAScript 6 создает привязки к переменным, функциям и классам, доступные только для чтения, а не просто ссылки на оригинальные привязки, как в случае с обычными переменными. Хотя модуль, импортирующий привязку, не сможет изменить ее значение, модуль, экспортирующий ее, сможет. Например, представьте, что вам потребовалось использовать следующий модуль:

```
export var name = "Nicholas";
export function setName(newName) {
  name = newName;
}
```

Импортировав эти две привязки, вы сможете изменить значение `name` вызовом `setName()`:

```
import { name, setName } from "./example.js";

console.log(name);    // "Nicholas"
setName("Greg");
console.log(name);    // "Greg"

name = "Nicholas";    // вызовет ошибку
```

Вызов `setName("Greg")` вернется обратно в модуль, откуда была экспортирована функция `setName()`, и выполнится там, записав в `name` строку `"Greg"`. Обратите внимание, что это изменение автоматически отразится на импортированной привязке `name`. Причина такого поведения в том, что `name` — это локальное имя для экспортированного идентификатора `name`. Привязка `name`, используемая в этом коде, и привязка `name`, используемая в модуле, это не одно и то же.

Переименование экспортируемых и импортируемых привязок

Иногда бывает нежелательно использовать оригинальное имя переменной, функции или класса, импортируемого из модуля. К счастью, есть возможность изменять имена при экспортировании и импортировании.

Рассмотрим первый случай. Представьте, что имеется функция, которую вы хотите экспортировать под другим именем. Вы можете использовать ключевое слово `as`, чтобы определить имя функции, под которым она будет известна за пределами модуля:

```
function sum(num1, num2) {
  return num1 + num2;
}
```

```
export { sum as add };
```

Здесь функция с *локальным именем* `sum()` экспортируется под именем `add()`. Это означает, что когда другой модуль будет импортировать эту функцию, он должен использовать имя `add`:

```
import { add } from "./example.js";
```

Чтобы импортировать функцию под другим именем, также можно воспользоваться ключевым словом `as`:

```
import { add as sum } from "./example.js";
console.log(typeof add);    // "undefined"
console.log(sum(1, 2));     // 3
```

Этот код импортирует функцию `add()` и использует *импортируемое имя* для ее переименования в `sum()` (локальное имя в данном контексте). Изменение локального имени функции в инструкции `import` приведет к тому, что идентификатор `add()` в данном модуле будет недоступен, несмотря на то что модуль импортирует функцию `add()`.

Значения по умолчанию в модулях

Синтаксис модулей поддерживает возможность экспортирования и импортирования значений по умолчанию, потому что этот шаблон широко распространен в других системах модулей, таких как CommonJS (еще одна спецификация для использования JavaScript за пределами браузеров). *Значение по умолчанию* для модуля — это единственная переменная, функция или класс, определяемая с помощью ключевого слова `default`, и каждый модуль может иметь только одно экспортируемое значение по умолчанию. Попытка использовать ключевое слово `default` с несколькими экспортируемыми компонентами приводит к синтаксической ошибке.

Экспортирование значений по умолчанию

Ниже приводится простой пример использования ключевого слова `default`:

```
export default function(num1, num2) {  
    return num1 + num2;  
}
```

В качестве значения по умолчанию этот модуль экспортирует функцию. Ключевое слово `default` указывает, что это экспортируемое значение по умолчанию. Функция может быть анонимной, потому что ее представляет модуль.

Имеется также возможность экспортировать идентификатор как значение по умолчанию, поместив его после `default export`, например:

```
function sum(num1, num2) {  
    return num1 + num2;  
}
```

```
export default sum;
```

В этом примере сначала определяется функция `sum()`, а затем она экспортируется как значение по умолчанию модуля. Этот прием может пригодиться, когда значение по умолчанию является вычисляемым.

Второй способ экспортирования идентификатора как значения по умолчанию основан на использовании синтаксиса переименования:

```
function sum(num1, num2) {  
    return num1 + num2;  
}
```

```
export { sum as default };
```

Идентификатор `default` в инструкции экспортирования с переименованием имеет специальное значение и указывает, что значение экспортируется как значение по умолчанию. Так как `default` в JavaScript является ключевым словом, его нельзя использовать в качестве имени переменной, функции или класса; однако его можно использовать как имя свойства. Таким образом, использование `default` для переименования в инструкции экспорта — это специальный случай, позволяющий отличить экспортируемое значение по умолчанию от значений не по умолчанию. Этот синтаксис может пригодиться, чтобы перечислить несколько экспортируемых компонентов в одной инструкции `export`, включая значение по умолчанию.

Импортирование значений по умолчанию

Значение по умолчанию можно импортировать из модуля с помощью следующего синтаксиса:

```
// импортировать значение по умолчанию
import sum from "./example.js";

console.log(sum(1, 2)); // 3
```

Эта инструкция `import` импортирует значение по умолчанию из модуля *example.js*. Обратите внимание на отсутствие фигурных скобок, которые используются для перечисления импортируемых компонентов, не являющихся значениями по умолчанию. Локальное имя `sum` в данном случае представляет функцию, экспортированную указанным модулем как значение по умолчанию. Этот синтаксис выглядит яснее, и создатели ECMAScript 6 ожидают, что он станет доминирующей формой импортирования в Веб, позволяющей использовать уже существующие объекты.

Импортировать одну или несколько привязок вместе с привязкой по умолчанию, экспортируемых другим модулем, можно в одной инструкции `import`. Например, допустим, что имеется следующий модуль:

```
export let color = "red";

export default function(num1, num2) {
  return num1 + num2;
}
```

Переменную `color` и функцию по умолчанию можно импортировать с помощью следующей инструкции `import`:

```
import sum, { color } from "./example.js";

console.log(sum(1, 2)); // 3
console.log(color);     // "red"
```

Локальное имя для значения по умолчанию отделяется запятой от других импортируемых имен, которые дополнительно заключены в фигурные скобки. Запомните, что значение по умолчанию должно следовать первым в инструкции `import`.

По аналогии с экспортированием импортируемые значения по умолчанию тоже можно переименовывать:

```
import { default as sum, color } from "./example.js";

console.log(sum(1, 2)); // 3
console.log(color);     // "red"
```

В этом примере экспортированное значение по умолчанию (`default`) переименовывается в `sum` и дополнительно импортируется привязка `color`. Этот пример является эквивалентом предыдущего примера.

Реэкспорт привязки

Рано или поздно вам может потребоваться повторно экспортировать какие-то привязки, импортированные модулем. Например, если вы создаете библиотеку, состоящую из нескольких маленьких модулей. Повторно экспортировать импортированное значение можно с использованием тех же шаблонов, что уже обсуждались в этой главе:


```
import { sum } from "./example.js";
export { sum }
```

Хотя это и работает, но ту же задачу можно решить единственной инструкцией:

```
export { sum } from "./example.js";
```

Эта форма инструкции `export` отыщет в указанном модуле объявление `sum` и экспортирует его. Конечно, также поддерживается возможность экспорта значения под другим именем:

```
export { sum as add } from "./example.js";
```

Здесь из модуля *example.js* импортируется функция `sum` и затем экспортируется под именем `add`.

Если потребуется экспортировать все, что экспортирует другой модуль, можно использовать шаблон `*`:

```
export * from "./example.js";
```

Эта инструкция экспортирует все, в том числе и значение по умолчанию, что может повлиять на экспортирование значений, принадлежащих данному модулю. Например, если *example.js* экспортирует значение по умолчанию, вы уже не сможете определить новое экспортируемое значение по умолчанию, используя этот синтаксис.

Импортирование без привязок

Некоторые модули могут ничего не экспортировать, зато могут изменять объекты в глобальной области видимости. Хотя переменные, функции и классы, объявленные на верхнем уровне модуля, не попадают в глобальную область видимости, это не означает, что модуль не имеет доступа к этой глобальной области видимости. Общедоступные определения встроенных объектов, таких как `Array` и `Object`, доступны внутри модуля, и изменения в этих объектах отразятся на других модулях.

Например, можно определить модуль, который добавит метод `pushAll()` во все массивы:

```
// модуль, ничего не экспортирующий и не импортирующий
Array.prototype.pushAll = function(items) {

    // аргумент items должен быть массивом
    if (!Array.isArray(items)) {
        throw new TypeError("Argument must be an array.");
    }

    // использовать встроенный метод push() и оператор расширения
    return this.push(...items);
};
```

Это вполне допустимый модуль, даже если он ничего не экспортирует и не импортирует. Этот код можно использовать и как модуль, и как сценарий. Поскольку он ничего не экспортирует, его можно выполнить с помощью упрощенной версии инструкции `import`, которая не импортирует никаких привязок:

```
import "./example.js";

let colors = ["red", "green", "blue"];
let items = [];

items.pushAll(colors);
```

Этот код импортирует и выполняет модуль, содержащий метод `pushAll()`, в результате чего `pushAll()` добавляется в прототип массивов. Это означает, что теперь метод `pushAll()` будет доступен для использования во всех массивах внутри данного модуля.

ПРИМЕЧАНИЕ

Вероятнее всего, импортирование без привязок будет использоваться в основном для создания полифиллов и расширений.

Загрузка модулей

Спецификация ECMAScript 6 определяет синтаксис модулей, но не определяет, как они должны загружаться. Отчасти этот недостаток спецификации объясняется стремлением обеспечить независимость от конкретных окружений. Вместо того чтобы пытаться определить единые инструменты для использования во всех окружениях JavaScript, ECMAScript 6 определяет только синтаксис и обобщает механизм загрузки как неопределенную внутреннюю операцию с названием `HostResolveImportedModule`. Разработчики веб-браузеров и Node.js сами решают, как реализовать операцию `HostResolveImportedModule`, чтобы она лучше соответствовала их окружениям.

Использование модулей в веб-браузерах

Даже до появления ECMAScript 6 веб-браузеры поддерживали много способов подключения JavaScript к веб-приложениям, в том числе:

- загрузка файлов с кодом на JavaScript с помощью элемента `<script>`, атрибут `src` которого определяет место, откуда должен загружаться код;
- непосредственное встраивание кода на JavaScript в элементы `<script>` без атрибута `src`;
- загрузка файлов с кодом на JavaScript для его запуска в виде фоновых потоков выполнения (Web Worker или Service Worker).

Для полноценной поддержки модулей разработчики веб-браузеров должны были обновить все эти механизмы. Конкретные детали полностью описаны в спецификации HTML, и я кратко опишу их в следующих разделах.

Использование модулей в элементах `<script>`

По умолчанию элемент `<script>` загружает файлы JavaScript как сценарии, а не как модули. Это происходит, если атрибут `type` отсутствует или определяет тип содержимого как код на JavaScript (например, `"text/javascript"`). Такой элемент `<script>` может выполнить встроенный код или загрузить файл, местоположение которого определяется атрибутом `src`. Для поддержки модулей в набор допустимых значений атрибута `type` была добавлена строка `"module"`. Если присвоить атрибуту `type` значение `"module"`, браузер загрузит встроенный код или код в файле, на который ссылается атрибут `src`, как модуль, а не как сценарий. Например:

```
<!-- загрузить модуль из файла JavaScript -->
<script type="module" src="module.js"></script>

<!-- подключить встроенный код как модуль -->
<script type="module">

import { sum } from "./example.js";

let result = sum(1, 2);

</script>
```

Первый элемент `<script>` в этом примере загружает внешний файл модуля с помощью атрибута `src`. Единственное отличие между загрузкой модуля и сценария заключается в наличии атрибута `type="module"`. Второй элемент `<script>` содержит код модуля, встроенный непосредственно в веб-страницу. Переменная `result` не попадет в глобальную область видимости, потому что она объявлена внутри модуля (согласно определению элемента `<script>`), и поэтому не будет добавлена как свойство объекта `window`.

Как видите, подключение модулей к веб-страницам осуществляется очень просто и напоминает подключение сценариев. Однако сама процедура загрузки модулей имеет некоторые отличия.

ПРИМЕЧАНИЕ

Возможно, вы обратили внимание, что строка `"module"` не является определением типа содержимого, как тип `"text/javascript"`. Файлы с модулями JavaScript имеют тот же тип содержимого, что и другие файлы с кодом на JavaScript, поэтому их нельзя отличить по одному только типу содержимого. Кроме того, браузеры игнорируют элементы `<script>`, в которых атрибут `type` имеет недопустимое значение, поэтому браузеры, не поддерживающие модули, будут автоматически игнорировать строку `<script type="module">`, которая обеспечивает хорошую обратную совместимость.

Последовательность загрузки модулей в веб-браузерах

Модули отличаются от сценариев тем, что могут использовать инструкцию `import`, чтобы определить, какие еще файлы нужно загрузить для их правильной работы. Для поддержки этой функциональности элементы `<script type="module">` всегда действуют так, как если бы имели атрибут `defer`.

Атрибут `defer` можно не использовать для загрузки файлов сценариев, но он всегда применяется при загрузке файлов модулей. Загрузка файла модуля начинается, как только парсер HTML встретит элемент `<script type="module">` с атрибутом `src`, но его выполнение откладывается до момента завершения анализа документа. Кроме того, модули выполняются в том порядке, в каком они появляются в HTML-файле. Это означает, что модуль, указанный в первом элементе `<script type="module">`, гарантированно будет выполнен перед вторым, даже если второй модуль оформлен как встроенный код, а не в виде файла, на который ссылается атрибут `src`. Например:

```
<!-- этот модуль выполнится первым -->
<script type="module" src="module1.js"></script>

<!-- этот модуль выполнится вторым -->
<script type="module">
import { sum } from "./example.js";

let result = sum(1, 2);
```

```
</script>
```

```
<!-- этот модуль выполнится третьим -->
```

```
<script type="module" src="module2.js"></script>
```

Эти три элемента `<script>` выполняются в порядке их следования, то есть модуль *module1.js* гарантированно выполнится перед встроенным модулем, а встроенный модуль гарантированно выполнится перед модулем *module2.js*.

Каждый модуль может импортировать один или несколько других модулей, что несколько усложняет ситуацию. По этой причине сначала выполняется анализ модулей, чтобы выявить все инструкции `import`. Для каждой инструкции `import` запускается процедура загрузки (из сети или из кэша), но ни один из модулей не выполняется, пока не будут загружены и выполнены все импортированные ими ресурсы.

Все модули — и те, что явно указаны в элементах `<script type="module">`, и те, что перечислены в инструкциях `import`, — загружаются по порядку. В данном примере полная последовательность загрузки выглядит так:

1. Загружается и анализируется *module1.js*.
2. Рекурсивно загружаются и анализируются ресурсы, импортируемые модулем *module1.js*.
3. Анализируется модуль во встроенном коде.
4. Рекурсивно загружаются и анализируются ресурсы, импортируемые модулем во встроенном коде.
5. Загружается и анализируется *module2.js*.
6. Рекурсивно загружаются и анализируются ресурсы, импортируемые модулем *module2.js*.

После загрузки модулей их выполнение откладывается до завершения анализа документа. Когда парсинг документа закончится, выполняются следующие действия:

1. Рекурсивно выполняются ресурсы, импортируемые модулем *module1.js*.
2. Выполняется модуль *module1.js*.
3. Рекурсивно выполняются ресурсы, импортируемые модулем во встроенном коде.
4. Выполняется модуль во встроенном коде.
5. Рекурсивно выполняются ресурсы, импортируемые модулем *module2.js*.
6. Выполняется модуль *module2.js*.

Обратите внимание, что в отношении модуля во встроенном коде выполняются те же действия, что и для двух других, за исключением того, что его код не требуется предварительно загружать. В остальном последовательность загрузки импортируемых ресурсов и выполнения модулей остается той же.

ПРИМЕЧАНИЕ

Атрибут `defer` в элементах `<script type="module">` игнорируется, потому что они и так действуют, как если бы этот атрибут был указан.

Асинхронная загрузка модулей в веб-браузерах

Возможно, вы уже знакомы с атрибутом `async` в элементе `<script>`. Когда атрибут `async` применяется к сценариям, файлы сценариев выполняются сразу после загрузки и анализа. Однако порядок выполнения сценариев с атрибутом `async` может не совпадать с порядком их следования в документе. Сценарии всегда выполняются сразу после завершения загрузки, не ожидая, пока закончится парсинг документа.

Атрибут `async` можно также применять и к модулям. Если `async` присутствует в `<script type="module">`, модуль выполняется, подобно сценарию, с той лишь разницей, что перед выполнением модуля производится загрузка и выполнение всех импортируемых им ресурсов. Это гарантирует, что все ресурсы, необходимые модулю, будут загружены до его выполнения; однако вы не можете гарантировать, *когда* модуль будет выполнен. Взгляните на следующий фрагмент:

```
<!-- порядок выполнения этих модулей не гарантируется -->
<script type="module" async src="module1.js"></script>
<script type="module" async src="module2.js"></script>
```

В данном примере два модуля загружаются асинхронно. Просто взглянув на этот код, нельзя сказать, какой из них выполнится первым. Если загрузка модуля *module1.js* завершится раньше (включая все импортируемые им ресурсы), он выполнится первым. Если первым загрузится модуль *module2.js*, значит, первым выполнится он.

Загрузка модулей в фоновые потоки выполнения

Фоновые потоки выполнения, такие как Web Workers и Service Workers, выполняют код на JavaScript вне контекста веб-страницы. Создание нового потока выполнения начинается с создания нового экземпляра `Worker` (или другого класса) с последующей передачей ему местоположения файла JavaScript. По умолчанию механизм загрузки загружает файлы как сценарии, например:

```
// загрузить script.js как сценарий
let worker = new Worker("script.js");
```

Для поддержки загрузки модулей разработчики стандарта HTML добавили второй аргумент в эти конструкторы — объект со свойством `type`, которое по умолчанию получает значение `"script"`. Чтобы загрузить модуль, в свойстве `type` следует указать значение `"module"`:

```
// загрузить module.js как модуль
let worker = new Worker("module.js", { type: "module" });
```

Этот код загрузит *module.js* как модуль, а не как сценарий, потому что во втором аргументе конструктору передается объект со свойством `type`, имеющим значение `"module"`. (Свойство `type` имитирует атрибут `type` элемента `<script>`, позволяющий отличать модули и сценарии.) Вторым аргументом поддерживаются всеми механизмами фоновых потоков выполнения в веб-браузерах.

Модули в фоновых потоках выполняются в основном так же, как сценарии, кроме пары исключений. Во-первых, сценарии могут загружаться в фоновые потоки только из того же домена, откуда была загружена веб-страница, создавшая эти потоки, но к модулям применяется менее строгое ограничение. В отношении модулей, передаваемых непосредственно в фоновые потоки, действует то же самое ограничение, но они могут загружать файлы, имеющие соответствующие заголовки CORS (Cross-Origin Resource Sharing — совместное использование ресурсов между разными источниками). Во-вторых, сценарий, загруженный в фоновый поток, может использовать метод `self.importScripts()`, чтобы загрузить дополнительные сценарии, но в модулях вызов `self.importScripts()` всегда вызывает ошибку, потому что модули должны для этих целей использовать инструкцию `import`.

Разрешение спецификаторов модулей в браузерах

Все примеры в этой главе, которые демонстрировались до сих пор, использовали относительный путь (как в строке `./example.js`) в качестве спецификатора модуля. Браузеры требуют, чтобы спецификаторы модулей были определены в одном из следующих форматов:

- начинался с `/`, чтобы поиск начинался с корневого каталога;
- начинался с `./`, чтобы поиск начинался с текущего каталога;
- начинался с `../`, чтобы поиск начинался с родительского каталога;
- имел формат URL.

Например, допустим, что имеется файл модуля, находящийся по адресу `https://www.example.com/modules/module.js` и содержащий следующий код:

```
// импортирует модуль по адресу https://www.example.com/modules/example1.js
import { first } from "./example1.js";
```

```
// импортирует модуль по адресу https://www.example.com/example2.js
import { second } from "../example2.js";
```

```
// импортирует модуль по адресу https://www.example.com/example3.js
import { third } from "/example3.js";
```

```
// импортирует модуль по адресу https://www2.example.com/example4.js
import { fourth } from "https://www2.example.com/example4.js";
```

Все спецификаторы модулей в этом примере считаются допустимыми в браузерах, включая полный адрес URL в последней строке. (Вам нужно лишь убедиться, что на сервере `www2.example.com` правильно настроены заголовки CORS, разрешающие загрузку из другого домена.) Это единственные спецификаторы форматов, которые браузеры способны разрешать по умолчанию, однако пока еще не законченная спецификация загрузчиков модулей будет предусматривать также другие форматы.

А пока некоторые спецификаторы, имеющие вполне обычный вид, фактически считаются браузерами недопустимыми, и их применение вызывает ошибку, например:

```
// недопустимый – не начинается с /, ./ или ../
import { first } from "example.js";
```

```
// недопустимый – не начинается с /, ./ или ../
import { second } from "example/index.js";
```

Ни один из этих модулей не будет загружен браузером. Оба спецификатора имеют недопустимый формат (отсутствуют требуемые символы в начале), даже при том, что оба прекрасно будут обработаны, если указать их в атрибуте `src` тега `<script>`. Такое различие в поведении между `<script>` и `import` было допущено намеренно.

В заключение

Спецификация ECMAScript 6 добавила в язык поддержку модулей с целью дать возможность упаковки и инкапсуляции функциональных возможностей. Модули действуют иначе, чем сценарии, — их переменные, функции и классы, объявленные на верхнем уровне, не попадают в глобальную область видимости, а ссылка `this` внутри модуля имеет значение `undefined`. Для достижения такого поведения загрузка модулей производится в другом режиме.

Чтобы открыть доступ к любым функциональным возможностям модуля, их необходимо экспортировать. Экспортироваться могут все компоненты — переменные, функции и классы. Кроме того, каждый модуль может экспортировать что-то одно как значение по умолчанию. Другие модули могут импортировать все или некоторые экспортированные имена. Эти имена, однако, интерпретируются, как если бы были объявлены с помощью инструкции `let`, и действуют подобно блочным привязкам, которые не допускают возможности переопределения в том же модуле.

Модули могут ничего не экспортировать, если их цель — выполнить некоторые манипуляции в глобальной области видимости. Фактически импорт функциональных возможностей из таких модулей происходит без создания каких-либо привязок в области видимости модуля.

Так как модули должны выполняться в специальном режиме, в браузерах была реализована поддержка версии тега `<script type="module">`, сообщающей, что исходный файл или встроенный код должен выполняться как модуль. Тег `<script type="module">` загружает файлы модулей, как если бы имел атрибут `defer`. Кроме того, модули выполняются в порядке их перечисления во встраиваемом документе и после того, как документ будет загружен и проанализирован.

Мелкие изменения в ECMAScript 6

Помимо крупных изменений, описанных в этой книге, спецификация ECMAScript внесла также несколько мелких изменений, усовершенствующих язык JavaScript. Среди них улучшенные приемы работы с целыми числами, новые математические методы, поддержка идентификаторов с символами Юникода и формализованно свойство `__proto__`. Обо всех этих новшествах рассказывается в данном приложении.

Новые приемы работы с целыми числами

Для представления целых и вещественных чисел в JavaScript используется система кодирования IEEE 754, которая вот уже много лет вызывает большую неразбериху. Разработчики языка приложили максимум усилий, чтобы прикладным программистам не приходилось задумываться о деталях представления чисел, но иногда проблемы все же возникают. В ECMAScript 6 предпринята еще одна попытка решить эти проблемы за счет упрощения идентификации и обработки чисел.

Идентификация целых чисел

В ECMAScript 6 появился новый метод `Number.isInteger()`, который может определить, является ли указанное значение целым числом. Несмотря на то что в JavaScript используется формат IEEE 754 для представления обоих типов чисел, вещественные и целые числа хранятся по-разному. Работа метода `Number.isInteger()` основана на этих различиях в способах хранения, и когда в вызов этого метода передается значение, по его внутреннему представлению движок JavaScript определяет, является ли оно целым числом. В результате числа, которые выглядят как вещественные, в действительности могут храниться как целые, и для них `Number.isInteger()` будет возвращать `true`. Например:

```
console.log(Number.isInteger(25));      // true
console.log(Number.isInteger(25.0));    // true
console.log(Number.isInteger(25.1));    // false
```

В этом примере `Number.isInteger()` вернул `true` для чисел 25 и 25.0, несмотря на то что второе выглядит как вещественное число. Простое добавление десятичной точки еще не означает, что JavaScript автоматически будет интерпретировать это число как вещественное. Поскольку в действительности число 25.0 — это просто 25, оно хранится как целое. Однако число 25.1 хранится как вещественное, потому что имеет дробную часть.

Безопасные целые числа

Формат IEEE 754 предусматривает точное представление целых чисел только в диапазоне от -2^{53} до 2^{53} , а вне этого «безопасного» диапазона одинаковые двоичные представления повторно используются для представления нескольких числовых значений. Следовательно, JavaScript может уверенно представлять целые числа только в диапазоне IEEE 754, а для чисел вне его могут наблюдаться проблемы. Например, взгляните на следующий фрагмент:

```
console.log(Math.pow(2, 53));           // 9007199254740992
console.log(Math.pow(2, 53) + 1);       // 9007199254740992
```

Здесь нет опечатки, действительно два разных числа в JavaScript имеют одинаковое целочисленное представление. Эффект становится тем заметнее, чем дальше число выходит за пределы безопасного диапазона.

В ECMAScript 6 появился метод `Number.isSafeInteger()`, идентифицирующий числа, имеющие точное представление. Также были добавлены свойства `Number.MAX_SAFE_INTEGER` и `Number.MIN_SAFE_INTEGER` для представления верхней и нижней границ безопасного диапазона целых чисел соответственно. Возвращая `true`, метод `Number.isSafeInteger()` гарантирует, что значение является целым числом и попадает в безопасный диапазон целочисленных значений, например:

```
var inside = Number.MAX_SAFE_INTEGER,
    outside = inside + 1;

console.log(Number.isInteger(inside));           // true
console.log(Number.isSafeInteger(inside));       // true

console.log(Number.isInteger(outside));          // true
console.log(Number.isSafeInteger(outside));      // false
```

Число `inside` — наибольшее целое число в безопасном диапазоне, поэтому для него методы `Number.isInteger()` и `Number.isSafeInteger()` возвращают `true`. Число `outside` — первое сомнительное целочисленное значение; оно не считается безопасным, хотя и определяется как целое.

В большинстве случаев в арифметических операциях или операциях сравнения вы предпочтете использовать только безопасные целые числа, поэтому использование `Number.isSafeInteger()` на этапе проверки допустимости ввода может оказаться отличной идеей.

Новые математические методы

Расширение области применения JavaScript на игры и графику, подтолкнувшее разработчиков ECMAScript 6 к включению в JavaScript типизированных массивов, также потребовало повышения эффективности реализации многих математических операций в движке JavaScript. Но стратегии оптимизации, такие как `asm.js`, которые поддерживают подмножество языка JavaScript и увеличивают производительность, требуют больше информации для выполнения вычислений максимально быстрым способом. Например, знание того, как интерпретировать числа — как 32-разрядные целые или 64-разрядные вещественные, — имеет большое значение для аппаратных операций, которые выполняются намного быстрее, чем программные.

В результате в ECMAScript 6 добавили несколько новых методов в объект `Math` с целью ускорения типичных математических вычислений. Увеличение скорости вычислений также способствует увеличению общей скорости выполнения приложений, производящих большие объемы вычислений, к которым относятся и программы обработки графики. Новые методы перечислены в табл. А.1.

Таблица А.1. Новые методы объекта Math, добавленные в ECMAScript 6

Метод	Возвращает
Math.acosh(x)	Гиперболический арккосинус x
Math.asinh(x)	Гиперболический арксинус x
Math.atanh(x)	Гиперболический арктангенс x
Math.cbrt(x)	Кубический корень из x
Math.clz32(x)	Число старших нулевых разрядов в 32-разрядном целочисленном представлении x
Math.cosh(x)	Гиперболический косинус x
Math.expml(x)	Результат вычитания 1 из экспоненты x
Math.fround(x)	Ближайшее к x вещественное число одиночной точности
Math.hypot(...values)	Квадратный корень из суммы квадратов аргументов
Math.imul(x, y)	Результат умножения двух аргументов как 32-разрядных целых чисел
Math.log1p(x)	Натуральный логарифм из суммы 1 + x
Math.log2(x)	Логарифм по основанию 2 из x
Math.log10(x)	Логарифм по основанию 10 из x
Math.sign(x)	-1, если x – отрицательное число; 0, если x равно +0 или -0; 1, если x – положительное число
Math.sinh(x)	Гиперболический синус x
Math.tanh(x)	Гиперболический тангенс x
Math.trunc(x)	Целая часть (просто отбрасывает дробную часть из вещественных чисел)

Подробное описание новых методов и их особенностей далеко выходит за рамки данной книги. Но если в вашем приложении потребуется производить типовые математические вычисления, обязательно проверьте перечень новых методов в объекте **Math**, прежде чем браться за реализацию собственных.

Идентификаторы с символами Юникода

Спецификация ECMAScript 6 предлагает более полную поддержку Юникода, чем предыдущие версии JavaScript, и это также сказалось на диапазоне символов, которые можно использовать в идентификаторах. В ECMAScript 5 допускается использовать в идентификаторах экранированные последовательности Юникода. Например:

```
// допустимо в ECMAScript 5 и 6
var \u0061 = "abc";

console.log(\u0061);    // "abc"

// эквивалентно вызову:
console.log(a);          // "abc"
```

В этом примере после инструкции **var** для доступа к переменной можно использовать идентификатор **\u0061** или **a**. В качестве идентификаторов в ECMAScript 6 допускается также использовать экранированные последовательности кодовых пунктов Юникода, например:

```
// допустимо в ECMAScript 5 и 6
var \u{61} = "abc";

console.log(\u{61});    // "abc"
```

```
// эквивалентно вызову:  
console.log(a);           // "abc"
```

В этом примере экранированная последовательность `\u0061` была заменена эквивалентным кодовым пунктом. В остальном этот код действует точно так же, как предыдущий.

Кроме того, ECMAScript 6 формально определяет допустимые идентификаторы в терминах рекомендации № 31 в стандарте Юникода (Unicode Standard Annex #31) «Unicode Identifier and Pattern Syntax» (<http://unicode.org/reports/tr31/>), включающей следующие правила:

- Первый символ должен быть `$`, `_` или любым символом Юникода с унаследованным базовым свойством `ID_Start`.
- Последующие символы могут быть `$`, `_`, `\u200c` (разъединитель нулевой ширины), `\u200d` (соединитель нулевой ширины) или любым другим символом Юникода с унаследованным базовым свойством `ID_Continue`.

Унаследованные базовые свойства `ID_Start` и `ID_Continue` определены в рекомендации «Unicode Identifier and Pattern Syntax» и служат для идентификации символов, которые могут использоваться в идентификаторах, таких как имена переменных и доменные имена. Это требование не является специфическим для JavaScript.

Формализованное свойство `__proto__`

Еще до завершения работы над спецификацией ECMAScript 5 некоторые движки JavaScript уже реализовали нестандартное свойство с именем `__proto__`, которое можно использовать для чтения и записи внутреннего свойства `[[Prototype]]`. Фактически свойство `__proto__` было ранним предшественником методов `Object.getPrototypeOf()` и `Object.setPrototypeOf()`. Ожидать, что все движки JavaScript удалят это свойство, было нереально (свойство `__proto__` используется многими популярными библиотеками JavaScript), поэтому спецификация ECMAScript 6 формализовала поведение `__proto__`. Но формализация была добавлена в стандарт ECMA-262, приложение B, вместе со следующим предупреждением:

- Эти особенности не входят в ядро языка ECMAScript. Программисты не должны использовать эти особенности или предполагать их наличие при разработке нового кода на ECMAScript. Создателям реализаций ECMAScript не рекомендуется добавлять эти особенности, если эти реализации не используются в веб-браузерах или не требуется обеспечить совместимость с устаревшим кодом на ECMAScript, с которым могут столкнуться веббраузеры.

Вместо этого свойства спецификация ECMAScript рекомендует использовать методы `Object.getPrototypeOf()` и `Object.setPrototypeOf()`, потому что `__proto__` имеет следующие характеристики:

- В литералах объектов свойство `__proto__` можно указать только один раз. Если указать его дважды, это вызовет ошибку. Это единственное свойство литералов объектов, имеющее такое ограничение.
- Вычисляемая форма `["__proto__"]` действует подобно обычному свойству и не ссылается на текущий прототип объекта. К этой форме применяются все требования, предъявляемые к свойствам литералов объектов, в отличие от не-вычисляемой формы, которая имеет некоторые исключения.

Несмотря на то что спецификация не рекомендует пользоваться свойством `__proto__`, она определяет его не совсем обычно, на чем следует остановиться подробнее. В движках ECMAScript 6 свойство `Object.prototype.__proto__` определено как свойство с методами доступа, метод `get` которого вызывает `Object.getPrototypeOf()`, а метод `set` вызывает `Object.setPrototypeOf()`. Поэтому единственное существенное отличие между использованием `__proto__` и метода `Object.getPrototypeOf()` или `Object.setPrototypeOf()` заключается в том, что `__proto__` позволяет устанавливать прототип литерала объекта непосредственно. Вот как это работает:

```
let person = {
  getGreeting() {
    return "Hello";
  }
};

let dog = {
  getGreeting() {
    return "Woof";
  }
};

// прототипом является person
let friend = {
  __proto__: person
};

console.log(friend.getGreeting());           // "Hello"
console.log(Object.getPrototypeOf(friend) === person); // true
console.log(friend.__proto__ === person);    // true

// установить dog как прототип
friend.__proto__ = dog;
console.log(friend.getGreeting());           // "Woof"
console.log(friend.__proto__ === dog);       // true
console.log(Object.getPrototypeOf(friend) === dog); // true
```

Вместо вызова `Object.create()` для создания объекта `friend` в этом примере определяется стандартный литерал объекта, в котором присваивается значение свойству `__proto__`. С другой стороны, когда объект создается вызовом метода `Object.create()`, необходимо указывать полные дескрипторы, описывающие дополнительные свойства объекта.

Введение в ECMAScript 7 (2016)

Работа над ECMAScript 6 заняла несколько лет, и в результате в техническом комитете TC-39 пришли к выводу, что такая продолжительность процесса разработки совершенно неприемлема. Поэтому было решено перейти на годовой цикл выпуска изменений, чтобы гарантировать более быстрое внедрение в язык новых особенностей.

Более частые выпуски подразумевают меньшее количество нововведений в каждой последующей редакции ECMAScript, чем в ECMAScript 6. Чтобы обозначить переход на новый цикл, новые версии спецификации решено нумеровать не номером редакции, а номером года публикации. В результате редакция ECMAScript 6 теперь также известна как ECMAScript 2015, а ECMAScript 7 формально называется ECMAScript 2016. Комитет TC-39 предполагает использовать систему именования с номером года во всех будущих редакциях ECMAScript.

Работа над редакцией ECMAScript 2016 была завершена в марте 2016 года и включает только три дополнения к языку: новый математический оператор, новый метод массивов и новую синтаксическую ошибку. Все три новшества рассматриваются далее в этом приложении.

Оператор возведения в степень

Единственным изменением в синтаксисе JavaScript, предусмотренным в ECMAScript 2016, является *оператор возведения в степень*, выполняющий одноименную математическую операцию. В JavaScript уже имеется метод `Math.pow()`, выполняющий возведение в степень, но JavaScript оставался одним из немногих языков, требующих использования метода вместо формального оператора. Кроме того, некоторые разработчики утверждают, что оператор проще читается в коде.

Оператор возведения в степень имеет форму двух звездочек (`**`): левый операнд используется как основание, а правый — как степень. Например:

```
let result = 5 ** 2;

console.log(result);           // 25
console.log(result === Math.pow(5, 2)); // true
```

Этот пример вычисляет выражение 5^2 , результат которого равен 25. При желании те же вычисления все еще можно выполнять с помощью метода `Math.pow()`.

Порядок операций

Оператор возведения в степень имеет высший приоритет из всех двухместных операторов в JavaScript (унарные операторы имеют более высокий приоритет, чем `**`). Это означает, что он

выполняется первым в любом сложном выражении, например:

```
let result = 2 * 5 ** 2;
console.log(result); // 50
```

Здесь сначала будет найден результат 5^2 , затем полученное значение умножается на 2. Конечный результат получится равным 50.

Ограничения операндов

Оператор возведения в степень накладывает некоторые необычные ограничения, отсутствующие в других операторах. Левый операнд не может быть выражением с унарным оператором, кроме `++` и `-`. Например, следующий пример вызовет синтаксическую ошибку:

```
// синтаксическая ошибка
let result = -5 ** 2;
```

Выражение `-5` в данном примере расценивается как синтаксическая ошибка, потому что возникает неоднозначность в определении порядка выполнения операций. Должен ли унарный оператор `-` применяться к числу 5 или к результату выражения `5**2`? Запрет использования унарных выражений слева от оператора возведения в степень устраняет эту неоднозначность. Чтобы ясно обозначить свои намерения, следует заключить в круглые скобки `-5` или `5 ** 2`, как показано ниже:

```
// правильно
let result1 = -(5 ** 2); // результат равен -25
```

```
// тоже правильно
let result2 = (-5) ** 2; // результат равен 25
```

Если заключить в круглые скобки выражение, унарный оператор `-` будет применен ко всему выражению. Если заключить в круглые скобки `-5`, интерпретатор поймет, что во вторую степень требуется возвести число `5`.

Выражения с операторами `++` и `-` слева от оператора возведения в степень не требуется заключать в скобки, потому что поведение обоих операторов ясно определено как направленное на их операнды. Префиксный оператор `++` или `-` изменяет свой операнд перед выполнением любой другой операции, а постфиксные версии ничего не изменяют, пока все выражение не будет вычислено. В обоих случаях эти операторы не вызывают ошибок при использовании слева от оператора возведения в степень, например:

```
let num1 = 2,
    num2 = 2;

console.log(++num1 ** 2); // 9
console.log(num1);       // 3

console.log(num2-- ** 2); // 4
console.log(num2);       // 1
```

В этом примере значение `num1` увеличивается перед выполнением оператора возведения в степень, поэтому `num1` получает значение 3, и в результате операции получается 9. Переменная `num2` сохраняет значение 2 перед выполнением оператора возведения в степень и затем уменьшается до 1.

Метод `Array.prototype.includes()`

Возможно, вы помните, что ECMAScript 6 добавила метод `String.prototype.includes()` для проверки вхождения подстроки в строку. Первоначально предполагалось, что ECMAScript 6 добавит также метод `Array.prototype.includes()`, чтобы обеспечить единообразие функциональных возможностей строк и массивов. Но определение метода `Array.prototype.includes()` не было завершено до окончания работ над ECMAScript 6, поэтому метод `Array.prototype.includes()` был добавлен в редакции ECMAScript 2016.

Как используется метод `Array.prototype.includes()`

Метод `Array.prototype.includes()` принимает два аргумента: искомое значение и необязательный индекс, с которого должен начинаться поиск. При наличии второго аргумента `includes()` начинает поиск с указанного индекса. (По умолчанию поиск производится, начиная с индекса 0.) Если указанное значение найдено в массиве, возвращается `true`, в противном случае возвращается `false`. Например:

```
let values = [1, 2, 3];
console.log(values.includes(1));      // true
console.log(values.includes(0));      // false

// начать поиск с индекса 2
console.log(values.includes(1, 2));   // false
```

Здесь вызов `values.includes()` возвращает `true` для значения 1 и `false` — для значения 0, потому что 0 отсутствует в массиве. Когда в вызов `values.includes()` передается второй аргумент, требующий начать поиск с индекса 2 (в котором хранится значение 3), он возвращает `false`, потому что 1 отсутствует в элементах с индексами между 2 и концом массива.

Сравнение значений

Для сравнения значений метод `includes()` использует оператор `===`, но с одним исключением: `NaN` считается равным `NaN`, даже если выражение `NaN === NaN` возвращает `false`. Этим метод `includes()` отличается от метода `indexOf()`, который использует оператор `===` без всяких исключений. Следующий пример демонстрирует это отличие:

```
let values = [1, NaN, 2];

console.log(values.indexOf(NaN));     // -1
console.log(values.includes(NaN));     // true
```

Метод `values.indexOf()` возвращает `-1` для `NaN`, даже если значение `NaN` присутствует в массиве `values`. Метод `values.includes()`, напротив, возвращает `true` для `NaN`, потому что он использует другой способ сравнения значений.

Когда требуется просто проверить присутствие значения в массиве и не нужно определять его индекс, я советую использовать `includes()` из-за различий в интерпретации значения `NaN` между `includes()` и `indexOf()`. Если требуется узнать индекс элемента, в котором хранится искомое значение, используйте метод `indexOf()`.

Еще одна особенность этой реализации состоит в том, что она считает равными `+0` и `-0`. В этом случае `indexOf()` и `includes()` показывают одинаковое поведение:

```
let values = [1, +0, 2];

console.log(values.indexOf(-0));      // 1
console.log(values.includes(-0));     // true
```

Здесь оба метода, `indexOf()` и `includes()`, обнаруживают `+0`, когда требуется найти `—0`, потому что оба значения считаются равными. Обратите внимание, что этим они отличаются от метода `Object.is()`, который считает `+0` и `-0` разными значениями.

Изменение области видимости функций в строгом режиме

Когда спецификацией ECMAScript 5 был введен строгий режим, язык был немного проще, чем после выхода ECMAScript 6, но ECMAScript 6 все еще позволяет включать строгий режим с помощью директивы `"use strict"`. Когда эта директива используется в глобальной области видимости, весь код выполняется в строгом режиме; когда эта директива используется внутри функции, в строгом режиме выполняется только тело этой функции. В последнем случае в версии ECMAScript 6 возникает проблема из-за того, что параметры могут определяться более сложными способами, в частности с привлечением операции деструктуризации и определением значений по умолчанию.

Чтобы понять суть проблемы, рассмотрим следующий пример:

```
function doSomething(first = this) {  
    "use strict";  
    return first;  
}
```

Здесь именованный параметр `first` получает значение по умолчанию `this`. Можно было бы предположить, что `first` получит значение `undefined`, потому что в подобных случаях спецификация ECMAScript 6 требует от движков JavaScript интерпретировать параметры как выполняемые в строгом режиме. Но реализовать интерпретацию параметров в строгом режиме, когда директива `"use strict"` находится внутри функции, оказалось слишком сложно, потому что параметры со значениями по умолчанию сами могут быть функциями. Из-за этой сложности большинство движков JavaScript не реализуют эту особенность и оставляют в `this` ссылку на глобальный объект.

Эта сложность реализации объясняет, почему в ECMAScript 2016 было решено считать недопустимым использование деструктурированных параметров или параметров со значениями по умолчанию в функциях, где присутствует директива `"use strict"`. Если в теле функции используется директива `"use strict"`, она может иметь только *список простых параметров*, не предусматривающих деструктуризацию и не имеющих значений по умолчанию. Ниже приведены несколько примеров допустимого и недопустимого использования директивы:

```
// допустимо – используются только простые параметры
```

```
function okay(first, second) {  
    "use strict";  
    return first;  
}
```

```
// синтаксическая ошибка
```

```
function notOkay1(first, second=first) {  
    "use strict";  
    return first;  
}
```

```
// синтаксическая ошибка
```

```
function notOkay2({ first, second }) {  
    "use strict";  
    return first;  
}
```


Вы все еще можете использовать `"use strict"` со списками простых параметров, и именно поэтому `okay()` действует в полном соответствии с ожиданиями (то есть точно так же, как в ECMAScript 5). Функция `notOkay1()` вызывает синтаксическую ошибку, потому что согласно ECMAScript 2016 в функциях с параметрами, имеющими значения по умолчанию, нельзя использовать директиву `"use strict"`. Функция `notOkay2()` также вызывает синтаксическую ошибку, потому что директиву `"use strict"` нельзя использовать в функциях с деструктурированными параметрами.

В целом это изменение устраняет неоднозначность для прикладных программистов на JavaScript и сложность реализации для разработчиков движков JavaScript.