Введение в ECMAScript 6

Нараян Прасти

[PACKT] PUBLISHING

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Нараян Прасти

Введение в ECMAScript 6

Learning ECMAScript 6

Learn all the new ES6 features and be among the most prominent JavaScript developers who can write efficient JS programs as per the latest standards!

Narayan Prusty

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Введение в ECMAScript 6

Знакомьтесь с новыми функциями ES6 и присоединяйтесь к ведущим программистам JavaScript, пишущим эффективный код JS согласно последним стандартам!

Нараян Прасти

Москва, 2016

УДК 004.438ECMAScript 6
ББК 32.973.2
П70

П70 Нараян Прасти

Введение в ECMAScript 6. / пер. с англ. Рагимов Р. Н.– М.: ДМК Пресс, 2016. – 176 с.: ил.

ISBN 978-5-97060-392-5

Данная книга содержит пошаговые инструкции по использованию новых возможностей ECMAScript 6 вместо устаревших трюков и приемов программирования на JavaScript.

Книга начинается с знакомства со всеми встроенными объектами ES6 и описания создания итераторов ES6. Затем она расскажет, как писать асинхронный код с помощью ES6 в обычном стиле синхронного кода. Далее описывается использование программного интерфейса рефлексии Reflect API для исследования и изменения свойств объектов. Затем рассматривается создание прокси-объектов и их применение для перехвата и изменения поведения операций с объектами. Наконец, демонстрируются устаревшие методы модульного программирования, такие как IIFE, CommonJS, AMD и UMD, и сравниваются с модулями ES6, способными значительно увеличить производительность веб-сайтов.

Издание предназначено для программистов на JavaScript, обладающих базовыми навыками разработки, и желающим освоить новейшие возможности ECMAScript 6 для совершенствования своих программ, выполняемых на стороне клиента.

Original English language edition published by Published by Packt Publishing Ltd., Livery Place, 35 Livery Street, Birmingham B3 2PB, UK. Copyright © 2015 Packt Publishing. Russian-language edition copyright © 2016 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78588-444-3 (англ.)
ISBN 978-5-97060-392-5 (рус.)

Copyright © 2015 Packt Publishing
© Оформление, перевод на русский язык, ДМК Пресс, 2016

ОГЛАВЛЕНИЕ

Предисловие **10**

Об авторе **12**

О технических рецензентах **13**

Введение **16**

О чем рассказывается в этой книге 16

Что понадобится при чтении этой книги 17

Совместимость с ECMAScript 6 18

Запуск ECMAScript 6 в несовместимых реализациях 18

Кому адресована эта книга 19

Соглашения 19

Отзывы и пожелания 20

Скачивание исходного кода примеров 21

Нарушение авторских прав 21

Глава 1. Игры с синтаксисом **22**

Ключевое слово let 22

Объявление переменных с областью видимости в пределах функции ... 23

Объявление переменных с областью видимости в пределах блока 24

Повторное объявление переменных 25

Ключевое слово const 27

Область видимости констант 27

Ссылки на объекты при помощи констант 28

Значения параметров по умолчанию 29

Оператор расширения 30

Другие применения оператора расширения 31

Расширение нескольких массивов 32

Дополнительные параметры 32

Деструктивное присваивание 33

Деструктивное присваивание массивов 34

Деструктивное присваивание объектов 37

Стрелочные функции 39

Расширенные литералы объектов 41
Определение свойств 41
Определение методов 41
Вычисляемые имена свойств 42
Итоги 42

Глава 2. Знакомство с библиотекой 43

Работа с числами 43
Двоичное представление 44
Восьмеричное представление 44
Метод Number.isInteger(number) 45
Метод Number.isNaN(value) 45
Метод Number.isFinite(number) 46
Метод Number.isSafeInteger(number) 47
Свойство Number.EPSILON 48
Объект Math 49
Тригонометрические операции 49
Алгебраические операции 49
Прочие методы 50
Работа со строками 52
Управляющая последовательность для больших кодовых пунктов 53
Метод codePointAt(index) 53
Метод String.fromCodePoint(number1, ..., number 2) 53
Метод repeat(count) 54
Метод includes(string, index) 54
Метод startsWith(string, index) 54
Функция endsWith(string, index) 55
Нормализация 55
Шаблонные строки 57
Выражения 57
Массивы 60
Метод Array.from(iterable, mapFunc, this) 60
Метод Array.of(values...) 61
Метод fill(value, startIndex, endIndex) 61
Метод find(testingFunc, this) 62
Метод findIndex(testingFunc, this) 63
Метод copyWithin(targetIndex, startIndex, endIndex) 63
Методы entries(), keys() и values() 64
Коллекции 64
Буферные массивы 65
Типизированные массивы 67
Объект Set 68
Объект WeakSet 69
Объект Map 69
Объект WeakMap 70
Объект Object 71

Свойство __proto__ ... 71
Метод Object.is(value1, value2) ... 72
Метод Object.setPrototypeOf(object, prototype) ... 72
Метод Object.assign(targetObj, sourceObjs...) ... 72

Итоги ... 73

Глава 3. Использование итераторов ... 75

Символы в спецификации ES6 ... 75
Оператор typeof ... 76
Оператор new ... 76
Использование символов как ключей свойств ... 77
Метод Object.getOwnPropertySymbols() ... 77
Метод Symbol.for(string) ... 78
Встроенные символы ... 79

Протоколы итераций ... 79
Протокол итератора ... 79
Итерационный протокол ... 80

Генераторы ... 81
Метод return(value) ... 83
Метод throw(exception) ... 84
Ключевое слово yield* ... 84
Цикл for...of ... 85

Оптимизация хвостового вызова ... 86
Преобразование неконцевых вызовов в концевые вызовы ... 87

Итоги ... 88

Глава 4. Асинхронное программирование ... 89

Модель выполнения JavaScript ... 89

Разработка асинхронного кода ... 90
Асинхронный код, основанный на событиях ... 91
Асинхронный код, основанный на обратных вызовах ... 94

Объекты Promise в помощь ... 95
Конструктор Promise ... 96
Результат асинхронной операции ... 97
Метод then(onFulfilled, onRejected) ... 98
Метод catch(onRejected) ... 104
Метод Promise.resolve(value) ... 106
Метод Promise.reject(value) ... 107
Метод Promise.all(iterable) ... 107
Метод Promise.race(iterable) ... 108

Программные интерфейсы JavaScript, основанные на объектах Promise ... 109
Программный интерфейс состояния батареи ... 109
Программный интерфейс веб-криптографии ... 110

Итоги ... 111

Глава 5. Реализация Reflect API ... 112

Объект Reflect ... 112
Метод Reflect.apply(function, this, args) ... 113
Метод Reflect.construct(constructor, args, prototype) ... 113
Метод Reflect.defineProperty(object, property, descriptor) ... 114
Метод Reflect.deleteProperty(object, property) ... 117
Метод Reflect.enumerate(object) ... 118
Метод Reflect.get(object, property, this) ... 118
Метод Reflect.set(object, property, value, this) ... 119
Метод Reflect.getOwnPropertyDescriptor(object, property) ... 119
Метод Reflect.getPrototypeOf(object) ... 120
Метод Reflect.setPrototypeOf(object, prototype) ... 120
Метод Reflect.has(object, property) ... 121
Метод Reflect.isExtensible(object) ... 121
Метод Reflect.preventExtensions(object) ... 121
Метод Reflect.ownKeys(object) ... 122

Итоги ... 122

Глава 6. Использование прокси-объектов ... 123

Основы прокси-объектов ... 123
Терминология ... 124

Программный интерфейс Proxy API ... 124
Ловушки ... 125

Метод Proxy.revocable(target, handler) ... 137
Возможный сценарий использования ... 138

Использование прокси ... 138

Итоги ... 138

Глава 7. Прогулка по классам ... 139

Понимание объектно-ориентированной модели JavaScript ... 139
Типы данных JavaScript ... 140
Создание объектов ... 140
Понятие наследования ... 141
Конструкторы элементарных типов данных ... 145

Использование классов ... 146
Определение классов ... 147
Методы прототипа ... 149
Статические методы ... 152
Реализация наследования классов ... 152
Вычисляемые имена методов ... 154
Атрибуты свойств ... 155

ПРЕДИСЛОВИЕ

Прежде не было более подходящего времени для программирования на JavaScript. За последние несколько лет на наших глазах JavaScript прошел путь от языка, с которым никто не хотел иметь дела, до языка, которым все непременно желают овладеть. Разработка больших и сложных приложений для браузеров сегодня подталкивает развитие JavaScript, как никогда раньше. Фреймворки и полностью новые подходы к созданию приложений породили новые потребности в разработке на стороне клиента, и сообщество пришло к соглашению о необходимости их удовлетворения.

Спецификация ECMAScript 2015 или ES6, как ее обычно называют, наконец-то, привела язык в соответствие с нашими высокими требованиями. Значительными приобретениями является встроенная поддержка отложенных вычислений и модульной организации, но присутствуют также и небольшие дополнения, которые делают повседневную разработку более приятной. Познакомившись поближе с приемом деструктурирования объектов, вы будете удивляться, как прежде удавалось обходиться без него, а впервые воспользовавшись стрелочной функцией, вы никогда не захотите вновь использовать «function». С помощью функции «let» вы избавитесь от сложностей с областями видимости функций и от утечки переменных, и вам реже придется биться головой о стену.

ES6 – великолепный язык, совершивший невероятный скачок относительно ES5, а упорный труд многих членов сообщества сделал возможным его использование уже сегодня, не дожидаясь полной реализации в браузерах. Существуют инструменты преобразования кода ES6 в код, совместимый со спецификацией ES5, а это значит, что будущее уже здесь, его не надо ждать еще 5 лет, как это часто бывало с JavaScript.

Эта книга познакомит вас с наиболее полезными нововведениями в JavaScript и всей, доступной уже сейчас, функциональностью. Научит, как конструировать модульные приложения, используя встроенную систему модулей ES6, и как сделать код чище, лаконичнее и

более приятным для работы. Изучение нового стандарта является сложной задачей для любого разработчика, и я рад внести свой вклад, написав предисловие к книге, которая значительно упростит эту задачу.

Эта книга поможет вам сделать первые шаги в новом удивительном мире JavaScript, клиентских приложений и фреймворков, основанном на спецификации ES6. Я надеюсь, что дочитав эту книгу до конца, вы будете так же взволнованы, как и я.

Джек Франклин (Jack Franklin)
Программист JavaScript из GoCardless
@Jack_Franklin
http://www.jackfranklin.co.uk

ОБ АВТОРЕ

Нараян Прасти (Narayan Prusty) разработчик веб- и мобильных приложений. Специализируется на WordPress, HTML5, JavaScript, PHP, Solr и Cordova. Изучал эти технологии и создавал приложения с их помощью в течение многих лет.

Является основателем сайта QScutter.com, который проводит курсы по различным темам разработки приложений и имеет более 10 000 подписчиков по всему миру. Его личный блог http://www.QNimate.com один из самых популярных в рейтинге блогов Intel XDK и WordPress. Он также работает консультантом и внештатным разработчиком во многих компаниях по всему миру.

Посетите его личную страничку http://www.twitter.com/narayanprusty.

В первую очередь хочу выразить благодарность веб-сообществу. Без их блестящих успехов в документировании и щедрого обмена решениями, я не смог бы написать эту книгу. И, наконец, я благодарен моей семье за поддержку.

О ТЕХНИЧЕСКИХ РЕЦЕНЗЕНТАХ

Андреа Чиарелли (Andrea Chiarelli) имеет более чем 20 летний опыт работы инженером-программистом и техническим писателем. В своей профессиональной карьере использовал различные технологии: от C# до JavaScript, от ASP.NET до AngularJS, от REST до PhoneGap/Cordova.

Сотрудничал со многими электронными и печатными журналами, такими как *Computer Programming* и *ASP Today*, и был соавтором нескольких книг, изданных Wrox Press.

В настоящее время работает старшим инженером-программистом в итальянском офисе компании Apparound Inc., основанной в самом центре Силиконовой долины и занимающейся разработкой программ для мобильных устройств, а также является постоянным автором итальянского электронного журнала *HTML.it*, специализирующегося на веб-технологиях.

Филипп Реневье Гонен (Philippe Renevier Gonin) с 2005 года работает ассистентом профессора в Университете Софии Антиполис (Ницца, Франция). Преподает веб-технологии, программную инженерию (проектирование и разработку), и предмет «Взаимодействие человека с компьютером» (Human Computer Interaction, сокращено HCI). Как исследователь, Филипп занимается связями между интерактивным пользовательским дизайном (например, моделями пользователей и задач) и программным обеспечением (например, компонентной архитектурой и разработкой пользовательского интерфейса). В своих проектах часто использует программное обеспечение и инструменты Javascript, HMTL, CSS, Java (Android).

Доменико Лучиани (Domenico Luciani**),** увлеченный 22-летний программист. В настоящее время работает инженером-программистом в нескольких компаниях и обучается в университете Палермо.

С энтузиазмом занимается задачами распознавания образов. Отдает предпочтение компьютерной безопасности и пентестам, принимал участие в премиальных программах ряда компаний. Работал со многими технологиями: MongoDB, Node.js, PHP, PostgreSQL и C.

Пишет модули Node.js, которые публикует на сайте NPM. Не раз выступал в роли технического рецензента и в настоящее время изучает язык программирования GoLang, просто для удовольствия.

Является членом сообщества Maker и любит работать на своем одноплатным компьютере Raspberry Pi. Обожает писать код в текстовом редакторе Vim и управлять исходными текстами с помощью Git. Пишет тесты и участвует в проектах с открытым исходным кодом в Интернете.

В свободное время занимается бегом, любит паркур. Вы можете найти более подробную информацию о нем на сайте http://www.dlion.it.

Михир Моне (Mihir Mone), аспирант из Университета Монаша, Австралия. Окончил курс обучения по распределенным вычислениям, но сейчас занимается разработкой веб- и мобильных приложений

co.jp/), занимающемся созданием коммуникационной платформы реального времени и SDK под названием Konnect.

Как программист на JavaScript, разрабатывает JavaScript-библиотеки и ES6-совместимые веб-приложения на React.JS.

Он также участвует в разработке веб-фреймворка Xitrum (http://xitrum-framework.github.io/). В рамках этого проекта изучает функциональное программирование на языке Scala, создавая примеры приложений и занимаясь переводом документации.

Был техническим рецензентом книги *Learning Behavior-driven Development Javascript*, опубликованной издательством Packt Publishing.

Юрий Струмпфлохнер (Juri Strumpflohner) увлеченный разработчик, любит программировать, следит за последними тенденциями в веб-разработке и делится своими выводами с другими. Работает программным архитектором и техническим руководителем в компании поддержки электронного правительства, где отвечает за подготовку разработчиков, внедрение инноваций и контроль качества программного обеспечения.

В свободное время участвует в проектах с открытым исходным кодом, занимается рецензированием книг (подобных этой), переписывается в Twitter (@juristr) или пишет о последних новостях технологий веб-разработки в своем блоге на http://juristr.com. В настоящее время его особенно интересуют ES 2015 (ES6), AngularJS, React, Babel и все связанное с современной веб-разработкой.

Если Юрий не программирует, то учится или тренируется, занимаясь боевым искусством Йосейкан Будо (Yoseikan Budo), где достиг черного пояса второго дана. Вы можете связаться с ним в Twitter (@juristr) или посетить его блог http://juristr.com.

ВВЕДЕНИЕ

ECMAScript – это язык сценариев, стандартизированный организацией Ecma International в спецификациях ECMA-262 и ISO/IEC 16262. Языки сценариев, такие как JavaScript, JScript и ActionScript, являются надмножествами ECMAScript. Хотя в JavaScript, JScript, и ActionScript дают больше возможностей, чем в ECMAScript, определяя множество дополнительных объектов и методов, основные черты этих языков совпадают с чертами ECMAScript.

ECMAScript 6 является шестой версией и седьмой редакцией языка ECMAScript. Короткое его название «ES6».

Хотя JavaScript чрезвычайно мощный и гибкий язык, его часто критикуют за ненужную избыточность. Поэтому, разработчики часто используют абстракции JavaScript, такие как CoffeeScript и Typescript, имеющие упрощенный синтаксис, мощные функции, и компилирующиеся в JavaScript. Спецификация ES6 была введена для такого улучшения JavaScript, которое убедит разработчиков не обращаться к абстракциям или другим методам для написания качественного кода, увеличивающим время выполнения.

Особенности ES6 унаследованы от других популярных абстрагированных языков, таких как CoffeeScript. То есть, особенности языка ES6 схожи с особенностями других языков и не новы в мире программирования, в тоже время они являются новыми для JavaScript.

Эта книга содержит разъяснения, сопровождаемые примерами, всех особенностей новой версии ECMAScript – ECMAScript 6. Она посвящена реализации стандарта ECMAScript 6 в JavaScript. Все функции и примеры в этой книге работают во всех окружениях JavaScript, таких как браузеры, Node.js, Cordova и так далее.

О чем рассказывается в этой книге

Глава 1 «Игры с синтаксисом», описывает новые способы создания переменных и параметров функций. В этой главе подробно обсуждаются новые объекты и функции.

Глава 2 «Знакомство с библиотекой», знакомит с новыми методами существующих объектов, основанными на прототипах.

Глава 3 «Использование итераторов», описывает различные типы итераторов, доступных в ES6, и приемы создания пользовательских итераторов. Она также рассматривает оптимизацию хвостового вызова в ES6.

Глава 4 «Асинхронное программирование», рассказывает, как объекты Promise помогают упростить создание кода, выполняемого асинхронно.

Глава 5 «Реализация Reflect API», служит подробным руководством по отражению объектов в ES6.

Глава 6 «Использование прокси-объектов», объясняет, как определить нестандартное поведение основных операций над объектами с помощью прокси-объектов ES6.

Глава 7 «Прогулка по классам», описывает приемы объектно-ориентированного программирования с использованием классов ES6. Объясняет такие понятия, как наследование, конструкторы, абстракции, сокрытие информации и другие.

Глава 8 «Модульное программирование», рассматривает разные способы создания модулей в JavaScript. Охватывает технологи создания модулей, такие как IIFE, CommonJS, AMD, UMD и ES6.

Что понадобится при чтении этой книги

Если вы читаете эту книгу после того, как стандарт ES6 стал полностью поддерживаться всеми реализациями JavaScript, то вам не понадобится никакой специальной среды для тестирования примеров из нее. Вы сможете просто проверить примеры на любом выбранном вами движке.

Если вы читаете эту книгу, до того как ES6 стал полностью поддерживается всеми реализациями JavaScript, тогда для опробования фрагментов кода, приведенных в книге, можно использовать транскомпилятор ES6. Чтобы опробовать примеры в браузере, используйте шаблон веб-страницы, с присоединенным к ней транскомпилятором Traceur для преобразования кода ES6 в код ES5, для каждой загружаемой страницы:

```
<!doctype html>
<html>
```

```
<head>...</head>
<body>
...
<script src="traceur.js"></script>
<script src="bootstrap.js"></script>
<script type="module">
            // Поместите программный код на ES6 сюда
</script>
</body>
</html>
```

Загрузите сценарии `traceur.js` (https://google.github.io/traceur-compiler/bin/traceur.js) и `bootstrap.js` (https://google. github.io/traceur-compiler/src/bootstrap.js). Поместите их в папку, где находится HTML-файл, содержащий приведенный выше код.

В прилагаемых к книге файлах (пакет кода), транскомпилятор Traceur и упомянутые сценарии уже подключены. Эти файлы предназначены для опробования примеров в браузерах.

Опробовать примеры из *главы 4 «Асинхронное программирование»* можно только в браузере, так как они используют JQuery и AJAX. Здесь вам также понадобится веб-сервер.

Чтобы опробовать примеры из *главы 8 «Модульное программирование»* в браузере, вам понадобится веб-сервер. Но если вы используете среду Node.js, то веб-сервер вам не потребуется.

Совместимость с ECMAScript 6

Эта книга была написана до того, как все реализации JavaScript начали поддерживать все функции ES6.

Подготовка спецификаций ES6 уже завершена. Но еще не все реализации JavaScript поддерживают все особенности ES6. Я уверен, что к концу 2016 года все реализации JavaScript будут поддерживать ES6.

В рамках проекта Kangax создана таблица совместимости с ES6, где можете отслеживать поддержку различных функций ES6 разными реализациями JavaScript. Найти эту таблицу можно по адресу

Полифил (polyfill) – это фрагмент кода, обеспечивающий поддержку технологии, которую вы, как разработчик, ожидаете получить от реализации JavaScript. Помните, что полифилы поддерживают не все функции ES6. Список доступных полифилов и ссылок для их загрузки можно найти в на странице https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#ecmascript-6-harmony.

Транскомпилятор ES6 принимает исходный код на ES6 и выводит исходный код на ES5, совместимый со всеми реализациями JavaScript. Транскомпиляторы поддерживают больше возможностей, чем полифилы, но также не способны обеспечить поддержку всех функций ES6. Существуют множество разных транскомпиляторов, таких как Google Traceur (https://github.com/google/traceur-compiler), Google Caja (https://developers.google.com/caja/), Babel (https://babeljs.io/), Termi ES6 Transpiler (https://github.com/termi/es6-transpiler) и многие другие. Вы всегда должны транскомпилировать код ES6 в код ES5 перед подключением его к веб-страницам, а не каждый раз при загрузке страницы, чтобы не замедлять их загрузку.

Таким образом, с помощью транскомпиляторов и/или полифилов, можно начать писать код на ES6 до того, как все реализации начнут полностью поддерживать ES6, а реализации без поддержки ES6 уйдут в прошлое.

Кому адресована эта книга

Эта книга адресована всем, кто знаком с JavaScript. Вы не должны быть экспертом в JavaScript, чтобы разобраться в приведенных здесь сведениях. Эта книга поможет вам поднять ваше знание JavaScript на новый уровень.

Соглашения

В этой книге вы обнаружите несколько стилей оформления текста, которые разделяют различные виды информации. Ниже приводятся примеры этих стилей и поясняется их значение.

Фрагменты кода в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, адреса страниц в Интернете, пользовательский ввод и указатели Twitter будут выглядеть так: «Мы можем подключать другие контексты посредством использования директивы `include`».

Блоки программного кода оформляются, как показано ниже:

```
var a = 12; // доступна глобально
function myFunction()
{
  console.log(a);
  var b = 13; // доступна в пределах функции
  if(true)
  {
    var c = 14; // доступна в пределах функции
    console.log(b);
  }
  console.log(c);
}
myFunction();
```

Новые термины и **важные слова** будут выделены жирным шрифтом.

Предупреждения или важные сообщения будут выделены подобным образом.

Подсказки и советы будут выглядеть так.

Отзывы и пожелания

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

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

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе **Читателям – Файлы к книгам**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки всё равно случаются. Если вы найдёте ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдёте какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и «Packt» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, и помогающую нам предоставлять вам качественные материалы.

ГЛАВА 1. Игры с синтаксисом

Языку **JavaScript** недоставало некоторых синтаксических форм, присутствующих в ряде других языков программирования, таких как объявление констант, объявление переменных с областью видимостью внутри блока, извлечение данных из массивов, короткий синтаксис для объявления функций и так далее. Спецификация **ES6** добавляет много новых синтаксических конструкций в JavaScript, которые позволят программистам писать более компактный и эффективный код. Также ES6 дает возможность отказаться от применения грубых приемов, отрицательно влияющих на производительность и затрудняющих чтение кода. В этой главе описаны новые синтаксические возможности, предоставляемые ES6.

В этой главе мы рассмотрим:

- создание переменных с областью видимости в пределах блока с помощью ключевого слова `let`;
- создание констант с помощью ключевого слова `const`;
- оператор расширения и дополнительные параметры;
- извлечение данных из итерируемых объектов с помощью деструктивного присваивания;
- стрелочные функции;
- новый синтаксис создания свойств объектов.

Ключевое слово let

В ES6 для объявления переменных с областью видимости в пределах блока и возможностью инициализации их значений используется ключевое слово `let`. У программистов, привыкших писать на других языках программирования и начинающих осваивать JavaScript, часто возникают ошибки, связанные с их представлениями об области видимости переменных. Почти все популярные языки используют

единые правила определения области видимости переменных, но в JavaScript эти правила несколько отличаются из-за отсутствия переменных с областью видимости в пределах блока. В связи с этим увеличивается вероятность утечек памяти, усложняется чтение кода и его отладка.

Объявление переменных с областью видимости в пределах функции

Переменные JavaScript, объявленные с помощью ключевого слова `var` называются **переменными с областью видимости в пределах функции**. Переменные, объявленные вне функции, становятся глобальными, то есть доступными из любого места в сценарии. Соответственно, переменные, объявленные внутри функции, доступны только внутри функции, но не за ее пределами.

Ниже приводится пример создания переменных внутри и за пределами функции:

```
var a = 12; // доступна глобально
function myFunction()
{
  console.log(a);
  var b = 13; // доступна в пределах функции
  if(true)
  {
    var c = 14; // доступна в пределах функции
    console.log(b);
  }
  console.log(c);
}
myFunction();
```

Результаты выполнения примера:

```
12
13
14
```

Загрузка примеров кода

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе **Читателям – Файлы к книгам**.

Здесь, как видите, переменная `c` доступна в пределах всей функции, но в других языках программирования она была бы недоступна за пределами оператора `if`. Следовательно, программисты, привыкшие к другим языкам программирования, ожидают, что переменная `c` не определена вне пределов оператора `if`, но это не так. Спецификация ES6 вводит ключевое слово `let` для создания переменных с областью видимости в пределах блока.

Объявление переменных с областью видимости в пределах блока

Переменные, объявленные с использованием ключевого слова `let`, называются переменными с областью видимости в пределах блока. Такие переменные, объявленные вне функции, ведут себя как обычные переменные, то есть, они доступны глобально. Но когда переменные с областью видимости в пределах блока объявляются внутри блока, они доступны только в пределах этого блока (а также во всех вложенных в него блоках).

Блок может содержать в себе ни одного или сколько угодно операторов. Ограничивает блок пара фигурных скобок, то есть {}.

Возьмем предыдущий пример и, заменив ключевые слова `var` на `let`, посмотрим на результаты его выполнения:

```
let a = 12; // доступна глобально
function myFunction()
{
  console.log(a);
  let b = 13; // доступна в пределах функции
  if(true)
  {
    let c = 14; // доступна только в инструкции "if"
    console.log(b);
  }
  console.log(c);
}
myFunction();
```

Результаты выполнения примера:

```
12
13
Reference Error Exception
```

Теперь, результаты стали именно такими, какие ожидают программисты, привыкшие к другим языкам программирования.

Повторное объявление переменных

Если с помощью ключевого слова `var` второй раз объявить переменную с тем же именем (в той же области видимости), объявленная ранее переменная будет затерта. Например:

```
var a = 0;
var a = 1;
console.log(a);
function myFunction()
{
  var b = 2;
  var b = 3;
  console.log(b);
}
myFunction();
```

Результаты выполнения примера:

```
1
3
```

Ожидаемый результат. Но переменные, созданные с использованием ключевого слова `let`, ведут себя по-другому.

Если с помощью ключевого слова `let` второй раз объявить переменную с тем же именем (в той же области видимости), будет вызвано исключение `TypeError`. Например:

```
let a = 0;
let a = 1; // TypeError
function myFunction()
{
  let b = 2;
  let b = 3; // TypeError
  if(true)
  {
    let c = 4;
    let c = 5; // TypeError
  }
}
myFunction();
```

Если объявить переменную с именем, которое уже было использовано ранее в функции (или во вложенной функции) или во вну-

треннем блоке, с использованием ключевого слова `var` или `let`, соответственно, это будет другая переменная. Следующий пример иллюстрирует это правило:

```
var a = 1;
let b = 2;
function myFunction()
{
  var a = 3; // другая переменная
  let b = 4; // другая переменная
  if(true)
  {
    var a = 5; // затрет прежнее значение
    let b = 6; // другая переменная
    console.log(a);
    console.log(b);
  }
  console.log(a);
  console.log(b);
}
myFunction();
console.log(a);
console.log(b);
```

Результаты выполнения примера:

```
5
6
5
4
1
2
```

Ключевое слово `var` или ключевое слово `let`, что использовать?

В программном коде ES6 рекомендуется использовать ключевое слово `let`, потому что это улучшает характеристики использования памяти, уменьшает вероятность ошибок области видимости, предотвращает случайные ошибки и облегчает чтение кода. Но если вы привыкли к ключевому слову `var` и чувствуете себя комфортно, можете продолжать использовать его.

Вас может удивить, почему бы просто не изменить поведение ключевого слова `var` вместо ввода нового ключевого слова `let`? Причиной тому является необходимость обратной совместимости.

Ключевое слово const

Ключевое слово `const`, появившееся в ES6, используется для объявления переменных, доступных только для чтения, то есть переменных, значения которых нельзя изменить. До ES6, программисты, как правило, добавляли префиксы в имена переменных, которые должны были быть постоянным. Например, взгляните на следующий код:

```
var const_pi = 3.141;
var r = 2;
console.log(const_pi * r * r); // Выведет "12.564"
```

Величина `pi` должна быть неизменной. Но и с префиксом всегда остается шанс случайного изменения значения такой переменной где-то в программе, так как он не предполагает встроенной защиты значения `pi`. Префикса недостаточно для отслеживания значений постоянных переменных.

Поэтому, чтобы обеспечить встроенную защиту переменных с постоянными значениями, было введено ключевое слово `const`. Предыдущий пример можно переписать на языке ES6, как показано ниже:

```
const pi = 3.141;
var r = 2;
console.log(pi * r * r); // Выведет "12.564"
pi = 12; // Вызовет исключение read-only
```

Здесь при попытке изменить значение переменной `pi` будет возбуждено исключение.

Область видимости констант

Константы являются постоянными переменными с областью видимости в пределах блока, то есть, к ним применяются те же правила, что и к переменным, объявленным с помощью ключевого слова `let`. Следующий пример демонстрирует область видимости констант:

```
const a = 12; // доступна глобально
function myFunction()
{
  console.log(a);
  const b = 13; // доступна в пределах функции
  if(true)
  {
    const c = 14; // доступна только в инструкции "if"
    console.log(b);
  }
```

```
  console.log(c);
}
myFunction();
```

Результаты выполнения примера:

```
12
13
ReferenceError Exception
```

Из результатов выполнения видно, что константы, когда дело ка-

Значения параметров по умолчанию

В JavaScript нет возможность определить значения по умолчанию для параметров функций. Поэтому программисты, как правило, вынуждены сравнивать параметры со значением undefined (так как это и есть значение по умолчанию для отсутствующих параметров) и назначать им свои значения по умолчанию. Следующий пример демонстрирует, как это делается:

```
function myFunction(x, y, z)
{
  x = x === undefined ? 1 : x;
  y = y === undefined ? 2 : y;
  z = z === undefined ? 3 : z;
  console.log(x, y, z); // Выведет "6 7 3"
}
myFunction(6, 7);
```

ES6 предлагает новый синтаксис, упрощающий эту задачу. Следующий пример демонстрирует, как это же делается в ES6:

```
function myFunction(x = 1, y = 2, z = 3)
{
  console.log(x, y, z); // Выведет "6 7 3"
}
myFunction(6,7);
```

Кроме того, передача undefined воспринимается как отсутствие аргумента. Следующий пример демонстрирует это:

```
function myFunction(x = 1, y = 2, z = 3)
{
  console.log(x, y, z); // Выведет "1 7 9"
}
myFunction(undefined,7,9);
```

Значениями по умолчанию также могут быть выражения. Следующий пример демонстрирует это:

```
function myFunction(x = 1, y = 2, z = 3 + 5)
{
  console.log(x, y, z); // Выведет "6 7 8"
}
myFunction(6,7);
```

Оператор расширения

Оператор расширения записывается как «...». Он разбивает итерируемые объекты на отдельные значения.

Итерируемым называется объект, содержащий множество значений и реализующий протокол итераций ES6, что позволяет перебирать его значения. Массив – один из примеров встроенных итерируемых объектов.

Оператор расширения можно применить в любом месте, где ожидается несколько аргументов функции или несколько элементов (для массива литералов).

Оператор расширения обычно используется для переноса значений итерируемого объекта в аргументы функции. Давайте рассмотрим пример разделения массива на аргументы функции.

До появления ES6, чтобы передать значения из массива в аргументы функции, программисты применяли метод `apply()`. Например:

```
function myFunction(a, b)
{
  return a + b;
}
var data = [1, 4];
var result = myFunction.apply(null, data);
console.log(result); // Выведет "5"
```

Здесь метод `apply` принимает массив, извлекает из него значения, передает их в виде отдельных аргументов функции и затем вызывает ее.

ES6 обеспечивает более простой способ, основанный на использовании оператора расширения. Например:

```
function myFunction(a, b)
{
  return a + b;
}
let data = [1, 4];
let result = myFunction(...data);
console.log(result); // Выведет "5"
```

Во время выполнения, перед вызовом функции `myFunction`, интерпретатор JavaScript заменит `...data` выражением `1,4`:

```
let result = myFunction(...data);
```

Предыдущий код примет вид:

```
let result = myFunction(1,4);
```

После этого будет вызвана функция.

Оператор расширения не вызывает метод `apply()`. Среда выполнения JavaScript разбивает массив с помощью протоколов итераций, не имеющих ничего общего с методом `apply()`, но приводящих к тому же результату.

Другие применения оператора расширения

Оператор расширения не ограничивается только разбиением итерируемых объектов на аргументы функций, он может быть использован везде, где требуется несколько элементов. Так что он имеет множество применений. Давайте рассмотрим некоторые другие случаи использования оператора расширения для массивов.

Определение значений массива как части другого массива

Оператор расширения можно использовать для передачи значений из одного массива в другой. Следующий пример, демонстрирует, как сделать значения существующего массива частью другого массива при его создании.

```
let array1 = [2,3,4];
let array2 = [1, ...array1, 5, 6, 7];
console.log(array2); // Выведет "1, 2, 3, 4, 5, 6, 7"
```

Здесь строка:

```
let array2 = [1, ...array1, 5, 6, 7];
```

будет заменена строкой:

```
let array2 = [1, 2, 3, 4, 5, 6, 7];
```

Копирование значений из одного массива в другой

Иногда требуется скопировать значения из одного массива в конец другого массива.

До появления ES6 это делалось так:

```
var array1 = [2,3,4];
var array2 = [1];
Array.prototype.push.apply(array2, array1);
console.log(array2); // Выведет "1, 2, 3, 4"
```

Но в ES6 появился более простой способ сделать то же самое:

```
let array1 = [2,3,4];
let array2 = [1];
array2.push(...array1);
console.log(array2); // Выведет "1, 2, 3, 4"
```

Здесь метод `push` принимает в качестве аргументов последовательность переменных и добавляет их в конец массива, для которого он был вызван.

Здесь строка:

```
array2.push(...array1);
```

будет заменена строкой:

```
array2.push(2, 3, 4);
```

Расширение нескольких массивов

Имеется возможность расширить сразу несколько массивов в однострочном выражении. Например:

```
let array1 = [1];
let array2 = [2];
let array3 = [...array1, ...array2, ...[3, 4]];// несколько массивов
let array4 = [5];
function myFunction(a, b, c, d, e)
{
  return a+b+c+d+e;
}
let result = myFunction(...array3, ...array4); // несколько массивов
console.log(result); // Выведет "15"
```

Дополнительные параметры

Дополнительные, или необязательные параметры также записываются как «...». Последний параметр функции с префиксом «...» называется дополнительным параметром. Дополнительный параметр – это массив, содержащий остальные параметры функции, когда

количество ее аргументов превышает количество именованных параметров.

Дополнительный параметр используется для передачи переменного числа аргументов функции.

До появления ES6, чтобы получить дополнительные аргументы, переданные функции, программисты использовали объект `arguments`. Объект `arguments` не является массивом, но поддерживает некоторые интерфейсы, делающие его похожим на массив.

Следующий пример показывает, как использовать объект `arguments` для получения дополнительных аргументов:

```
function myFunction(a, b)
{
  var args = Array.prototype.slice.call(arguments, myFunction.length);
  console.log(args);
}
myFunction(1, 2, 3, 4, 5); // Выведет "3, 4, 5"
```

В ES6 это может быть сделано куда проще и элегантнее с помощью дополнительного параметра. Вот пример использования дополнительного параметра:

```
function myFunction(a, b, ...args)
{
  console.log(args); // Выведет "3, 4, 5"
}
myFunction(1, 2, 3, 4, 5);
```

Объект `arguments` не является массивом. Поэтому, чтобы применить к нему операции для массивов, необходимо сначала преобразовать его в массив. Дополнительный параметр, напротив, изначально является массивом, что упрощает работу с ним.

Как действует «...»?

Когда интерпретатор встречает многоточие «...», он обрабатывает его либо как оператор расширения, либо как дополнительный параметр, в зависимости от того, где и как он применен.

Деструктивное присваивание

Деструктивное присваивание – это выражение, позволяющее назначать переменным значения массивов или свойства итерируемых объ-

ектов с помощью синтаксиса, похожего на синтаксис создания массивов или объектов, соответственно.

Деструктивное присваивание упрощает извлечение данных из массивов или итерируемых объектов, обеспечивая более короткий синтаксис. Деструктивное присваивание уже присутствует в таких языках программирования, как Perl и Python, и выполняется так же, как и везде.

Есть два вида выражений деструктивного присваивания: для массивов и для объектов. Рассмотрим подробно каждый из них.

Деструктивное присваивание массивов

Деструктивное присваивание массивов используется для извлечения значений итерируемого объекта и присваивания их переменным. Оно названо *деструктивным присваиванием массивов*, потому что похоже на выражение создания литерала массива.

До появления ES6, программисты присваивали значения массива переменным следующим образом:

```
var myArray = [1, 2, 3];
var a = myArray[0];
var b = myArray[1];
var c = myArray[2];
```

Здесь последовательно извлекаются значения массива и присваиваются переменным `a`, `b` и `c`.

В ES6 то же самое можно сделать одной строкой, применив деструктивное присваивание массивов:

```
let myArray = [1, 2, 3];
let a, b, c;
[a, b, c] = myArray; // синтаксис деструктивного присваивания
                     // массива
```

Здесь `[a, b, c]` – это деструктивное выражение.

В левой части оператора деструктивного присваивания мы должны поместить переменные, которым хотим присвоить значения элементов массива, используя синтаксис, аналогичный синтаксису создания литерала массива. В правой части – массив (точнее, любой итерируемый объект), значения которого мы хотим извлечь.

Предыдущий пример можно сделать еще более коротким:

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

Здесь мы создали переменные в том же операторе и вместо переменной массива использовали литерал массива.

Если переменных меньше, чем элементов в массиве, учитываются только первые элементы.

Если в правой части деструктивного присваивания поместить неитерируемый объект, это вызовет исключение `TypeError`.

Игнорирование значений

Имеется также возможность игнорировать некоторые значения в итерируемом объекте. Следующий пример показывает, как это сделать:

```
let [a, , b] = [1, 2, 3];
console.log(a);
console.log(b);
```

Результаты выполнения примера:

```
1
3
```

Использование оператора расширения в деструктивном присваивании массивов

Для последней переменной деструктивного присваивания массива можно использовать оператор «...». В этом случае переменная будет преобразована в массив объектов с остальными значениями итерируемого объекта, если число других переменных меньше числа значений в итерируемом объекте.

Для понимания, как это делается, рассмотрим пример:

```
let [a, ...b] = [1, 2, 3, 4, 5, 6];
console.log(a);
console.log(Array.isArray(b));
console.log(b);
```

Результаты выполнения предыдущего примера приведены ниже:

```
1
true
2,3,4,5,6
```

Здесь переменная `b` преобразуется в массив, содержащий все остальные элементы массива справа.

Здесь многоточие «...» называется **оператором дополнения**.

С помощью оператора дополнения можно также проигнорировать значения. Вот пример, демонстрирующий это:

```
let [a, , ,...b] = [1, 2, 3, 4, 5, 6];
console.log(a);
console.log(b);
```

Результаты выполнения предыдущего примера:

```
1
4,5,6
```

Здесь мы проигнорировали значения 2, 3.

Значения по умолчанию для переменных

В деструктивном присваивании можно указать значения по умолчанию для переменных, которые будут использованы вместо undefined, если массив не содержит соответствующих им значений. Вот пример, демонстрирующий это:

```
let [a, b, c = 3] = [1, 2];
console.log(c); // Выведет "3"
```

Деструктивное присваивание вложенных массивов

Имеется также возможность извлечения значений и из многомерных массивов с последующим присваиванием переменным. Вот пример, демонстрирующий это:

```
let [a, b, [c, d]] = [1, 2, [3, 4]];
```

Использование деструктивного присваивания как параметра

Выражение деструктивного присваивания можно также использовать в качестве параметра функции, чтобы извлечь значения итерируемого объекта и передать их как аргументы в параметры функции. Следующий пример демонстрирует это:

```
function myFunction([a, b, c = 3])
{
  console.log(a, b, c); // Выведет "1 2 3"
}
myFunction([1, 2]);
```

Как было показано выше в этой главе, если передать значение undefined в качестве аргумента при вызове функции, JavaScript прове-

рит наличие значения параметра по умолчанию. Следовательно, в качестве значения по умолчанию можно указать массив, который будет использован, если аргумент получит значение undefined. Следующий пример демонстрирует это:

```
function myFunction([a, b, c = 3] = [1, 2, 3])
{
  console.log(a, b, c); // Выведет "1 2 3"
}
myFunction(undefined);
```

Здесь мы передаем undefined в качестве аргумента и массив [1, 2, 3], который затем был использован для извлечения значений.

Деструктивное присваивание объектов

Деструктивное присваивание объектов используется для выделения и присваивания их свойств переменным.

До появления ES6, программисты делали это следующим образом:

```
var object = {"name" : "John", "age" : 23};
var name = object.name;
var age = object.age;
```

В ES6 то же самое можно уместить в одну строку, применив деструктивное присваивание объектов:

```
let object = {"name" : "John", "age" : 23};
let name, age;
({name, age} = object); // синтаксис деструктивного присваивания
                        // объектов
```

В левую часть оператора деструктивного присваивания должны быть помешены переменные для присваивания свойств объекта. В правую часть – объект, свойства которого мы хотим извлечь, а затем, обернуть всю инструкцию круглыми скобками ().

Имена переменных должны совпадать с именами свойств объекта. Если потребуется использовать переменные с другими именами, это

Здесь мы определили пустой объект, как значение по умолчанию параметра, который будет использован, если функции будет передан аргумент undefined.

Стрелочные функции

ES6 предоставляет новый способ создания функций с помощью оператора =>. Такие функции называются **стрелочными**. Их определение имеет более компактный синтаксис, но все стрелочные функции являются анонимными.

Следующий пример демонстрирует создание стрелочной функции:

```
let circleArea = (pi, r) => {
  let area = pi * r * r;
  return area;
}
let result = circleArea(3.14, 3);
console.log(result); // Выведет "28.26"
```

Здесь переменная circleArea содержит ссылку на анонимную стрелочную функцию. Предыдущий фрагмент на языке ES5 выглядит так:

```
Var circleArea = function(pi, r) {
  var area = pi * r * r;
  return area;
}
var result = circleArea(3.14, 3);
console.log(result); // Выведет "28.26"
```

Если стрелочная функция содержит только один оператор, нет необходимости заключать ее в фигурные скобки {}. Например:

```
let circleArea = (pi, r) => pi * r * r;
let result = circleArea(3.14, 3);
console.log(result); // Выведет "28.26"
```

Когда фигурные скобки {} не используются, функция автоматически возвращает значение единственного оператора.

Значение ключевого слова this в стрелочных функциях

В стрелочных функциях ключевое слово this имеет такое же значение, как в содержащей ее области (глобальной или области функции, внутри которой определена стрелочная функция), а не ссылку

на объект контекста (то есть, на объект стрелочной функции), как это бывает в обычных функциях.

Рассмотрим следующий пример, чтобы понять разницу значений `this` в обычной и в стрелочной функциях:

```
var object = {
  f1: function(){
    console.log(this);
    var f2 = function(){ console.log(this); }
  f2();
  setTimeout(f2, 1000);
  }
}
object.f1();
```

Результаты выполнения предыдущего примера:

```
Object
Window
Window
```

Здесь, ключевое слово `this` внутри функции `f1` содержит ссылку на объект `object`, так как функция `f1` является его свойством. Но `this` внутри `f2` содержит ссылку на объект `window`, так как `f2` является свойством объекта `window`.

В стрелочных функциях `this` ведет себя иначе. Давайте, заменим в предыдущем примере обычные функции стрелочными и посмотрим на результат :

```
var object = {
  f1: () => {
    console.log(this);
    var f2 = () => { console.log(this); }
    f2();
    setTimeout(f2, 1000);
  }
}
object.f1();
```

Результаты выполнения предыдущего примера:

```
Window
Window
Window
```

Здесь, ключевое слово `this` внутри функции `f1` дублирует значение `this` глобального контекста, так как `f1` принадлежит глобально-

му контексту. Ключевое слово `this` внутри функции `f2` дублирует значение `this` функции `f1`, так как функция `f2` принадлежит области функции `f1`.

Другие различия между стрелочными и традиционными функциями

Стрелочные функции нельзя использовать как конструкторы объектов, к ним не может быть применен оператор `new`.

Помимо различий, касающихся значения `this` и оператора `new`, в остальном стрелочные и традиционные функции идентичны, потому что являются экземплярами одного и того же конструктора `Function`.

Расширенные литералы объектов

Спецификация ES6 добавила несколько новых синтаксических рас-

```
let object = {
  myFunction(){
   console.log("Hello World!!!"); // Выведет "Hello World!!!"
  }
}
object.myFunction();
```

Эта компактная форма записи функций позволяет использовать внутри них ключевое слово `super`, тогда как традиционные методы объектов не дают такой возможности. Подробнее об этом рассказывается далее в книге.

Вычисляемые имена свойств

Имена свойств, которые вычисляются во время выполнения, называются **вычисляемыми именами свойств**. Имя свойства является результатом вычисления выражения.

В ES5 вычисляемые свойства определяются следующим образом:

```
var object = {};
object["first"+"Name"] = "Eden";// "firstName" - имя свойства
// извлечь
console.log(object["first"+"Name"]); // Выведет "Eden"
```

Здесь, после создания объекта, мы добавляем в него новые свойства. Но в ES6 есть возможность добавлять свойства с вычисляемыми именами прямо при создании объектов. Например:

```
let object = {
  ["first" + "Name"]: "Eden",
};
// извлечь
console.log(object["first" + "Name"]); // Выведет "Eden"
```

Итоги

В этой главе мы узнали об областях видимости переменных, переменных только для чтения, расщеплении массивов на отдельные значения, передаче неопределенных параметров в функции, извлечении данных из объектов и массивов, стрелочных функциях и новом синтаксисе создания свойств объекта.

В следующей главе мы рассмотрим встроенные объекты, символы и новые свойства строк, массивов и объектов, появившиеся в ES6.

ГЛАВА 2. Знакомство с библиотекой

Спецификация **ES6** добавляет во встроенные объекты JavaScript много новых свойств и методов, облегчающих жизнь разработчикам. Целью этих новых возможностей является стремление помочь программистам отказаться от использования непереносимых и потенциально ошибочных приемов реализации различных операций, связанных с числами, строками и массивами. В этой главе мы рассмотрим весь новый функционал, добавленный спецификацией ES6 во встроенные объекты.

В этой главе будут описаны:

- новые свойства и методы объекта `Number`;
- двоичное и восьмеричное представление числовых констант;
- новые свойства и методы объекта `Math`;
- создание многострочных строк и новые методы объекта `String`;
- новые свойства и методы объекта `Array`;
- объекты `Map` и `Set`;
- использование буферных и типизированных массивов;
- новые свойства и методы объекта `Object`.

Работа с числами

Спецификация ES6 добавляет новые способы создания чисел и новые свойства в объект `Number` для облечения работы с числами. Спецификация ES6 расширяет объект `Number` для упрощения создания насыщенных математикой приложений и предотвращения ошибок, связанных с вводящими в заблуждение особенностями. Кроме того спецификация ES6 предоставляет новые способы реализации уже доступных в ES5 операций, например, восьмеричное представление числовых констант.

предлагает новый префикс `0o` для числовых констант в восьмеричном представлении.

Следующий пример демонстрирует его использование:

```
let a = 0o17;
let b = 15;
console.log(a === b);
console.log(a);
```

Результат выполнения:

```
true
15
```

Метод Number.isInteger(number)

Числа в JavaScript хранятся как 64-битные числа с плавающей точкой. То есть, целые числа в JavaScript тоже являются числами с плавающей точкой без дробной части или, точнее, десятичным числом с дробной частью, равной нулю.

В спецификации ES5 не предусмотрено встроенного метода проверки, является ли число целым. Спецификация ES6 добавляет новый метод `isInteger()` в объект `Number`, который принимает в качестве параметра число и возвращает `true` или `false`, в зависимости от того является ли число целым или нет.

Например:

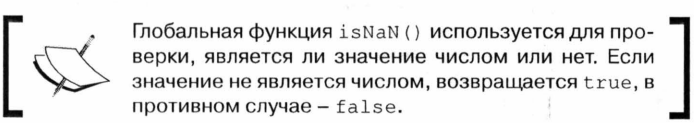
```
let a = 17.0;
let b = 1.2;
console.log(Number.isInteger(a));
console.log(Number.isInteger(b));
```

Результат выполнения:

```
true
false
```

Метод Number.isNaN(value)

В спецификации ES5 не предусмотрено встроенного метода проверки равенства переменной значению `NaN`.



Глобальная функция `isNaN()` используется для проверки, является ли значение числом или нет. Если значение не является числом, возвращается `true`, в противном случае – `false`.

Спецификация ES6 добавляет в объект `Number` новый метод `isNaN()` для проверки является ли значение числом или нет. Вот пример, демонстрирующий использование метода `Number.isNaN()`, а также иллюстрирующий его отличие от глобальной функции `IsNaN()`:

```
let a = "NaN";
let b = NaN;
let c = "hello";
let d = 12;
console.log(Number.isNaN(a));
console.log(Number.isNaN(b));
console.log(Number.isNaN(c));
console.log(Number.isNaN(d));
console.log(isNaN(a));
console.log(isNaN(b));
console.log(isNaN(c));
console.log(isNaN(d));
```

Результат выполнения примера:

```
false
true
false
false
true
true
true
false
```

Заметьте, что метод `Number.isNaN()` возвращает `true`, только когда ему передано значение `NaN`.

Вы можете спросить, почему бы просто не использовать оператор `==` или `===` вместо метода `Number.isNaN(value)`? Значение `NaN` является единственным значением, которое не равно самому себе, то есть, выражения `NaN==NaN` и `NaN===NaN` возвращают `false`.

Метод Number.isFinite(number)

В спецификации ES5 не предусмотрено встроенного метода проверки, является ли значение конечным числом.

Глобальная функция `isFinite()` принимает одно значение и проверяет, является ли оно конечным числом или нет. К сожалению, она также возвращает `true` для значений, которые преобразуются к типу `Number`.

Спецификация ES6 добавляет метод `Number.isFinite()`, в котором устранены недостатки функции `window.isFinite()`. Следующий пример, демонстрирует это:

```
console.log(isFinite(10));
console.log(isFinite(NaN));
console.log(isFinite(null));
console.log(isFinite([]));
console.log(Number.isFinite(10));
console.log(Number.isFinite(NaN));
console.log(Number.isFinite(null));
console.log(Number.isFinite([]));
```

Вот результат выполнения этого примера:

```
true
false
true
true
true
false
false
false
```

Метод Number.isSafeInteger(number)

Числа в JavaScript хранятся как 64-разрядные числа с плавающей точкой, в соответствии с международным стандартом IEEE 754. Этот формат использует для хранения числа 64 бита, где число (дробная часть) занимает биты с 0 по 51, экспонента – биты с 52 по 62, а знак числа – последний бит.

Таким образом, в JavaScript безопасными считаются целые числа, которые не будут округлены до других целых чисел, чтобы уместить в представление IEEE 754. Математически такими безопасными целыми числами являются числа от $-(2^{53}-1)$ до $(2^{53}-1)$.

Следующий пример демонстрирует это:

```
console.log(Number.isSafeInteger(156));
console.log(Number.isSafeInteger('1212'));
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER));
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1));
console.log(Number.isSafeInteger(Number.MIN_SAFE_INTEGER));
console.log(Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1));
```

Результат выполнения этого примера:

```
true
false
```

```
true
false
true
false
```

Здесь `Number.MAX_SAFE_INTEGER` и `Number.MIN_SAFE_INTEGER` являются константами, введенными в спецификацию ES6 и равными $(2^{53}-1)$ и $-(2^{53}-1)$ соответственно.

Свойство Number.EPSILON

JavaScript использует такое двоичное представление чисел с плавающей точкой, что некоторые числа, такие как 0.1, 0.2, 0.3 и другие, имеют погрешность. При выполнении числа, подобные 0.1, округляются до ближайшего числа в этом формате, что приводит к появлению небольшой ошибки.

Рассмотрим пример:

```
console.log(0.1 + 0.2 == 0.3);
console.log(0.9 - 0.8 == 0.1);
console.log(0.1 + 0.2);
console.log(0.9 - 0.8);
```

Результат выполнения кода этого примера:

```
false
false
0.30000000000000004
0.09999999999999998
```

В спецификацию ES6 введено свойство `Number.EPSILON`, имеющее значение, приблизительно равное 2^{-52}. Это значение представляет расчетную погрешность при сравнении чисел с плавающей запятой. Используя его, можно создать свою функцию сравнения чисел с плавающей запятой, игнорирующую минимальные ошибки округления.

Например:

```
functionepsilonEqual(a, b)
{
  return Math.abs(a - b) <Number.EPSILON;
}
console.log(epsilonEqual(0.1 + 0.2, 0.3));
console.log(epsilonEqual(0.9 - 0.8, 0.1));
```

Результат выполнения этого примера:

```
true
true
```

Здесь функция `epsilonEqual()` – это пользовательская функция сравнения двух значений на равенство. Теперь, мы получили именно то, что ожидали.

Больше узнать об арифметике чисел с плавающей точкой в JavaScript можно по адресу http://floating-point-gui.de/.

Объект Math

Спецификация ES6 добавляет много новых методов в объект `Math`, связанных с тригонометрией, алгеброй, и прочих. Это позволяет разработчикам пользоваться встроенными методами, а не внешними библиотеками. Встроенные методы лучше оптимизированы и обеспечивают большую точность.

Тригонометрические операции

Следующий пример демонстрирует все вновь добавленные в объект `Math` тригонометрические методы:

```
console.log(Math.sinh(0)); // гиперболический синус
console.log(Math.cosh(0)); // гиперболический косинус
console.log(Math.tanh(0)); // гиперболический тангенс
console.log(Math.asinh(0)); // обратный гиперболический синус
console.log(Math.acosh(1)); // обратный гиперболический косинус
console.log(Math.atanh(0)); // обратный гиперболический тангенс
console.log(Math.hypot(2, 2, 1));// Теорема Пифагора
```

Результат выполнения примера:

```
0
1
0
0
0
0
3
```

Алгебраические операции

Следующий пример демонстрирует все вновь добавленные в объект `Math` алгебраические методы:

```
console.log(Math.log2(16)); // логарифм по основанию 2
console.log(Math.log10(1000)); // логарифм по основанию 10
```

```
console.log(Math.log1p(0)); // то же, что и log(1 + value)
console.log(Math.expm1(0)); // функция, обратная Math.log1p()
console.log(Math.cbrt(8)); // корень кубический
```

Результат выполнения примера:

```
4
3
0
0
2
```

Прочие методы

Спецификация ES6 добавляет еще несколько методов в объект `Math`. Эти методы используются для преобразования и извлечения информации из чисел.

Функция Math.imul(number1, number2)

Функция `Math.imul()` принимает два 32-битных целых числа, перемножает их и возвращает младшие 32 бита результата умножения. Это единственный встроенный способ умножения 32-битных целых чисел в JavaScript.

Пример, демонстрирующий работу функции:

```
console.log(Math.imul(590, 5000000)); // умножение 32-битных целых
console.log(590 * 5000000); // умножение 64-битных вещественных
```

Результат выполнения примера:

```
-1344967296
2950000000
```

Когда произведение настолько велико, что не умещается в 32 бита, происходит потеря ведущих битов.

Функция Math.clz32(number)

Функция `Math.clz32()` возвращает число ведущих нулевых бит в 32-битном представлении числа.

Пример, демонстрирующий работу функции:

```
console.log(Math.clz32(7));
console.log(Math.clz32(1000));
console.log(Math.clz32(295000000));
```

Результат выполнения примера:

```
29
22
3
```

Функция Math.sign(number)

Функция `Math.sign()` возвращает знак числа, сообщающий, является ли число отрицательным, положительным или равно нулю.

Пример, демонстрирующий работу функции:

```
console.log(Math.sign(11));
console.log(Math.sign(-11));
console.log(Math.sign(0));
```

Результат выполнения примера:

```
1
-1
0
```

Из результатов выполнения предыдущего примера видно, что функция `Math.sign()` возвращает `1` если число положительное, `-1`, если число отрицательное и `0`, если число равно нулю.

Функция Math.trunc(number)

Функция `Math.trunc()` возвращает целую часть числа, отбрасывая дробную часть.

Пример, демонстрирующий работу функции:

```
console.log(Math.trunc(11.17));
console.log(Math.trunc(-1.112));
```

Результат выполнения примера:

```
11
-1
```

Функция Math.fround(number)

Функция `Math.fround()` округляет число до 32-битного значения с плавающей точкой.

Пример, демонстрирующий работу функции:

```
console.log(Math.fround(0));
console.log(Math.fround(1));
console.log(Math.fround(1.137));
console.log(Math.fround(1.5));
```

Результат выполнения примера:

```
0
1
1.1369999647140503
1.5
```

Работа со строками

Спецификация ES6 определяет новые способы создания строк и добавляет свойства в глобальный объект `String` и его экземпляры для облегчения работы со строками. Строкам в JavaScript не хватало функций и возможностей, имеющихся в других языках программирования, таких как Python и Ruby, теперь спецификация ES6 заполнила этот пробел.

Прежде чем перейти к рассмотрению новых функций со строками, давайте остановимся на внутренней кодировке JavaScript и управляющих последовательностях. В наборе символов Unicode каждый символ представлен десятичным числом, которое называется **кодовым пунктом** (code point). **Кодовой единицей** (code unit) называется постоянное число бит в памяти, используемое для хранения одного кодового пункта. Размер кодовой единицы определяется схемой кодирования. В схеме кодирования **UTF-8** кодовая единица имеет размер 8 бит, в схеме **UTF-16** кодовая единица имеет размер 16 бит. Если кодовые пункты не вписываются в кодовую единицу, происходит расщепление на несколько кодовых единиц, то есть, один символ будет представлен последовательностью из нескольких кодовых единиц.

Интерпретаторы JavaScript по умолчанию воспринимают исходный код JavaScript как последовательность кодовых единиц UTF-16. Если исходный код написан в схеме UTF-8, существуют различные пути оповестить интерпретатор JavaScript о необходимости воспринимать его как последовательность кодовых единиц UTF-8. Строки JavaScript всегда представляют собой последовательность кодовых пунктов UTF-16.

Любой символ Unicode с кодовым пунктом меньше 65536 можно записать в строке JavaScript или в исходном коде с помощью управляющей последовательности виде шестнадцатеричного значения его кодового пункта с префиксом `\u`. Управляющая последовательность всегда должна содержать шесть символов. За префиксом `\u` должны следовать ровно четыре символа. Если шестнадцатеричный код символа имеет длину в один, два или три символа, необходимо дополнить его ведущими нулями. Например:

```
var a = "\u0061\u0062\u0063";
console.log(a); // Выведет "abc"
```

Управляющая последовательность для больших кодовых пунктов

В ES5 для записи символа, занимающего при хранении более 16 бит, требуется использовать две управляющие последовательности Unicode. Например, чтобы добавить в строку кодовый пункт `\u1F691`, ее нужно записать так:

```
console.log("\uD83D\uDE91");
```

Пара последовательностей `\uD83D` и `\uDE91` в ней называются **суррогатной парой** (surrogate pair). Суррогатная пара состоит из двух символов Unicode, записанных последовательно для представления единственного другого символа.

В ES6 такой символ можно записать без суррогатной пары:

```
console.log("\u{1F691}");
```

Строка `\u1F691` при хранении идентична строке `\uD83D\uDE91`, то есть длина обеих строк равна 2.

Метод codePointAt(index)

Метод `codePointAt()` объекта `String` возвращает неотрицательное целое число, представляющее кодовый пункт символа для указанного индекса.

Следующий пример демонстрирует работу метода:

```
console.log("\uD83D\uDE91".codePointAt(1));
console.log("\u{1F691}".codePointAt(1));
console.log("hello".codePointAt(2));
```

Результат выполнения примера:

```
56977
56977
1080
```

Метод String.fromCodePoint(number1, ..., number 2)

Метод `fromCodePoint()` объекта `String` принимает последовательность кодовых пунктов и возвращает строку. Пример, демонстрирующий работу метода:

```
console.log(String.fromCodePoint(0x61, 0x62, 0x63));
console.log("\u0061\u0062 " == String.fromCodePoint(0x61, 0x62));
```

Результат выполнения примера:

```
abc
true
```

Метод repeat(count)

Метод `repeat()` строки конструирует и возвращает новую строку, содержащую заданное количество копий строки, для которой он был вызван, сцепленных вместе. Пример, демонстрирующий работу метода:

```
console.log("a".repeat(6)); // Выведет "aaaaaa"
```

Метод includes(string, index)

Метод `includes()` используется для поиска одной строки в другой и возвращает `true` или `false`, в зависимости от результата. Пример, демонстрирующий метод:

```
var str = "Hi, I am a JS Developer";
console.log(str.includes("JS")); // Выведет "true"
```

Метод принимает необязательный второй параметр с начальной позицией для поиска. Пример, демонстрирующий это:

```
var str = "Hi, I am a JS Developer";
console.log(str.includes("JS", 13)); // Выведет "false"
```

Метод startsWith(string, index)

Метод `startsWith()` используется для проверки, начинается ли строка с другой строки, возвращая `true` или `false`, в зависимости от результата. Пример, демонстрирующий метод:

```
var str = "Hi, I am a JS Developer";
console.log(str.startsWith('Hi, I am')); // Выведет "true"
```

Метод принимает необязательный второй параметр с начальной позицией в строке для проверки. Пример, демонстрирующий это:

```
var str = "Hi, I am a JS Developer";
console.log(str.startsWith('JS Developer', 11)); // Выведет "true"
```

Функция endsWith(string, index)

Метод `endsWith()` используется для проверки, заканчивается ли строка другой строкой, возвращая `true` или `false`, в зависимости от результата. Метод также принимает необязательный второй параметр с позицией в строке для поиска, которая откладывается от конца строки. Пример, демонстрирующий метод:

```
var str = "Hi, I am a JS Developer";
console.log(str.endsWith("JS Developer")); // Выведет "true"
console.log(str.endsWith("JS", 13)); // Выведет "true"
```

Нормализация

Нормализацией (normalization) называется процесс стандартизации кодовых пунктов без изменения состава строки.

Существуют разные формы нормализации: NFC, NFD, NFKC и NFKD.

Рассмотрим нормализацию строк Unicode на примере ее использования:

Изучение на примере

Есть множество символов Unicode, которые можно сохранить в 16 битах, и в то же время заданы с помощью суррогатной пары. Например, символ `'é'` можно определить двумя способами:

```
console.log("\u00E9"); // выведет 'é'
console.log("e\u0301"); // выведет 'é'
```

Проблема в том, что при применении оператора `==`, индексировании или при расчете длины вы получите неожиданный результат. Например:

```
var a = "\u00E9";
var b = "e\u0301";
console.log(a == b);
console.log(a.length);
console.log(b.length);
for(let i = 0; i<a.length; i++)
{
  console.log(a[i]);
}
for(let i = 0; i<b.length; i++)
{
  console.log(b[i]);
}
```

Результат выполнения примера:

```
false
1
2
é
é
```

Здесь обе строки отображаются одинаково, но при выполнении строковых операций мы получаем разные результаты.

Свойство `length` игнорирует суррогатные пары и воспринимает каждые 16 бит как один символ. Оператор == сравнивает двоичные разряды, поэтому он также игнорирует суррогатные пары. Оператор `[]` также основывается на 16-битном индексе, поэтому и он игнорирует суррогатные пары.

Чтобы решить эту проблему, следует преобразовать суррогатные пары в 16-битное представление символов. Этот процесс называется нормализацией. Для этого спецификация ES6 предоставляет функцию `normalize()`.Пример, демонстрирующий нормализацию:

```
var a = "\u00E9".normalize();
var b = "e\u0301".normalize();
console.log(a == b);
console.log(a.length);
console.log(b.length);
for(let i = 0; i<a.length; i++)
{
  console.log(a[i]);
}
for(let i = 0; i<b.length; i++)
{
  console.log(b[i]);
}
```

Результат выполнения примера:

```
true
1
1
é
é
```

Это уже ожидаемый результат. Функция `normalize()` возвращает нормализованную версию строки. По умолчанию `normalize()` использует форму NFC.

Нормализация применяется не только для замены суррогатных пар, существует много других областей ее применения.

Нормализованная версия строки не предназначена для отображения перед пользователем, она используется только для сравнения и поиска в строках.

Дополнительную информацию о нормализации строк Unicode и формах нормализации можно найти на странице http://www.unicode.org/reports/tr15/.

Шаблонные строки

Шаблонные строки – это просто новый литерал для создания строк, упрощающий различные операции. Они предоставляют, например, встроенные выражения, многострочные строки, интерполяцию, форматирование и тегирование (tagging) строк, и так далее. Шаблонные строки всегда обрабатываются и преобразуются в обычные строки JavaScript, следовательно, могут использоваться везде, где могут использоваться обычные строки

Шаблонные строки заключаются в обратные апострофы вместо одинарных или двойных кавычек. Пример простой шаблонной строки:

```
let str1 = `hello!!!`; // шаблонная строка
let str2 = "hello!!!";
console.log(str1 === str2); // выведет "true"
```

Выражения

В спецификации ES5 для включения выражений в обычные строки вы должны сделать примерно следующее:

```
var a = 20;
var b = 10;
var c = "JavaScript";
var str = "My age is " + (a + b) + " and I love " + c;
console.log(str);
```

Результат выполнения примера:

```
My age is 30 and I love JavaScript
```

В спецификации ES6 стало гораздо проще вставлять выражения в шаблонные строки. Шаблонные строки могут содержать в себе выражения. Выражения помещаются в заполнители, обозначаемые символом доллара и фигурными скобками, то есть, `${expressions}`. Интерпретатор вычисляет значения выражений заполнителей, и

полученный текст передается функции преобразования шаблонной строки в обычную строку. По умолчанию функция просто объединяет части в одну строку. Если для обработки частей строки используется пользовательская функция, шаблонная строка называется **тегированной шаблонной строкой** (tagged template string), а пользовательская функция называется **теговой функцией** (tag function).

Следующий пример демонстрирует, как встраивать выражения в шаблонные строки:

```
let a = 20;
let b = 10;
let c = "JavaScript";
let str = `My age is ${a+b} and I love ${c}`;
console.log(str);
```

Результат выполнения примера:

```
My age is 30 and I love JavaScript
```

А теперь создадим тегированную шаблонную строку, то есть, обработаем строку теговой функцией. Давайте реализуем теговую функцию, повторяющую действия функции по умолчанию:

```
let tag = function(strings, ...values)
{
  let result = "";
  for(let i = 0; i<strings.length; i++)
  {
    result += strings[i];
    if(i<values.length)
    {
      result += values[i];
    }
  }
  return result;
};
let a = 20;
let b = 10;
let c = "JavaScript";
let str = tag `My age is ${a+b} and I love ${c}`;
console.log(str);
```

Результат выполнения примера:

```
My age is 30 and I love JavaScript
```

В примере наша теговая функция названа `tag`, но ее можно назвать как угодно. Пользовательская функция принимает два параметра,

первый из которых является массивом строковых литералов из шаблонной строки, а второй – массивом результатов вычисления выражений. Второй параметр передается как несколько аргументов, поэтому используем дополнительный аргумент.

Многострочные строки

Шаблонные строки обеспечивают новый способ создания строк содержащих несколько строк текста.

В спецификации ES5 с этой целью необходимо использовать символ новой строки `\n`. Например:

```
console.log("1\n2\n3");
```

Результат выполнения примера:

```
1
2
3
```

Чтобы создать **многострочную** (multiline) строку в спецификации ES6, достаточно записать:

```
console.log(`1
2
3`);
```

Результат выполнения:

```
1
2
3
```

В примере выше мы просто включили в строку символы перехода на новую строку, там, где должны были находиться символы `\n`. При преобразовании шаблонной строки в обычную строку они будут преобразованы в символы `\n`.

Неформатированные строки

Неформатированной строкой является обычная строка, в которой управляющие последовательности не интерпретируются.

Создать неформатированную строку можно с помощью шаблонной строки. Получить неформатированную версию шаблонной строки можно с помощью теговой функции `String.raw`. Например:

```
let s = String.raw `xy\n${ 1 + 1 }z`;
console.log(s);
```

Результат выполнения:

```
xy\n2z
```

Здесь `\n` не интерпретируется как символ новой строки, замещающий два символа, а воспринимается буквально, как символы `\` и `n`. Длина переменной `s` будет равна `6`.

Если вы создаете теговую функцию и хотите вернуть неформатированную строку, используйте свойство `raw` первого аргумента. Свойство `raw` является массивом, который содержит неформатированные версии строк. Следующий пример демонстрирует это:

```
let tag = function(strings, ...values)
{
  return strings.raw[0]
};
let str = tag `Hello \n World!!!`;
console.log(str);
```

Результат выполнения:

```
Hello \n World!!!
```

Массивы

Спецификация ES6 добавляет новые свойства в глобальный объект `Array` и в его экземпляры для облегчения работы с массивами. Массивам в JavaScript не хватает функций и возможностей, имеющихся в других языках программирования, таких как Python и Ruby, спецификация ES6 исправляет этот недостаток.

Метод Array.from(iterable, mapFunc, this)

Метод `Array.from()` создает новый экземпляр массива из итерируемого объекта. Первым аргументом является ссылка на итерируемый объект. Второй аргумент является необязательным и может содержать ссылку на функцию обратного вызова (**функцию отображения**), которая вызывается для каждого элемента итерируемого объекта. Третий аргумент также является необязательным и представляет значение ключевого слова `this` внутри функции отображения.

Пример, демонстрирующий метод:

```
let str = "0123";
let obj = {number: 1};
```

```
let arr = Array.from(str, function(value){
  return parseInt(value) + this.number;
}, obj);
console.log(arr);
```

Результат выполнения:

```
1, 2, 3, 4
```

Метод Array.of(values...)

Метод `Array.of()` используется в качестве альтернативы конструктору объекта `Array` для создания массивов. Если передать конструктору `Array` только один числовой аргумент, он создаст пустой массив со свойством `length`, равным указанному числу, вместо массива с одним элементом, содержащим это число. Новый метод `Array.of()` решает эту проблему.

Например:

```
let arr1 = new Array(2);
let arr2 = new Array.of(2);
console.log(arr1[0], arr1.length);
console.log(arr2[0], arr2.length);
```

Результат выполнения:

```
undefined 2
2 1
```

Вы должны использовать метод `Array.of()` вместо конструктора объекта `Array`, когда динамически создаете новый экземпляр массива, то есть, когда тип его значений и количество элементов неизвестны.

Метод fill(value, startIndex, endIndex)

Метод `fill()` объекта `Array` заменяет все элементы массива от `startIndex` до `endIndex` (исключая `endIndex`) заданным значением. Имейте в виду, что аргументы `startIndex` и `endIndex` являются необязательными и если они отсутствуют, заданным значением заполняется весь массив. Если передать только аргумент `startIndex`, аргумент `endIndex` по умолчанию получит значение, равное длине массива минус 1.

Если аргумент `startIndex` меньше нуля, он интерпретируется как длина массива плюс `startIndex`. Если аргумент `endIndex` меньше нуля, он интерпретируется как длина массива плюс `endIndex`.

Например:

```
let arr1 = [1, 2, 3, 4];
let arr2 = [1, 2, 3, 4];
let arr3 = [1, 2, 3, 4];
let arr4 = [1, 2, 3, 4];
let arr5 = [1, 2, 3, 4];
arr1.fill(5);
arr2.fill(5, 1, 2);
arr3.fill(5, 1, 3);
arr4.fill(5, -3, 2);
arr5.fill(5, 0, -2);
console.log(arr1);
console.log(arr2);
console.log(arr3);
console.log(arr4);
console.log(arr5);
```

```
      return true;
    }
  }, x);
  console.log(result); // Выведет "12"
```

Метод findIndex(testingFunc, this)

Метод findIndex() действует подобно методу find(), но возвращает индекс элемента массива удовлетворяющего условию, а не сам элемент.

```
let x = 12;
let arr = [11, 12, 13];
let result = arr.findIndex(function(value, index, array){
  if(value == this)
  {
    return true;
  }
}, x);
console.log(result); // Выведет "1"
```

Метод copyWithin(targetIndex, startIndex, endIndex)

Метод copyWithin() объекта Array копирует последовательность значений массива в другое место в этом же массиве.

Метод copyWithin() принимает три аргумента: первый аргумент представляет индекс места назначения, куда будут скопированы элементы, второй – представляет индекс первого копируемого элемента, и третий – представляет индекс конечного копируемого элемента.

```
console.log(arr1);
console.log(arr2);
console.log(arr3);
console.log(arr4);
```

Результат выполнения:

```
1,3,4,4,5
2,3,4,5,5
1,4,5,4,5
1,4,3,4,5
```

Методы entries(), keys() и values()

Метод `entries()` массива возвращает итерируемый объект, содержащий пары ключ/значение для каждого индекса массива. Метод `keys()` массива возвращает итерируемый объект, содержащий ключи для всех индексов массива. Аналогично, метод `values()` массива возвращает итерируемый объект, содержащий значения элементов массива.

Итерируемый объект, возвращаемый методом `entries()`, хранит пары ключ/значение в форме массивов.

Итерируемые объекты, возвращаемые этими функциями, не являются массивами.

Пример, демонстрирующий использование методов:

```
let arr = ['a', 'b', 'c'];
let entries = arr.entries();
let keys = arr.keys();
let values = arr.values();
console.log(...entries);
console.log(...keys);
console.log(...values);
```

Результат выполнения:

```
0,a 1,b 2,c
0 1 2
a b c
```

Коллекции

Коллекцией является любой объект, который хранит множество элементов, как единое целое. Спецификация ES6 вводит новые объекты коллекций для обеспечения более эффективных способов хранения и организации данных.

В спецификации ES5 единственной коллекцией является массив. Спецификация ES6 добавляет буферные массивы, типизированные массивы, объекты `Set` и `Map`, которые также являются коллекциями.

Рассмотрим различные коллекции, предоставляемые спецификацией ES6.

Буферные массивы

Элементами массивов могут быть значения любых типов, таких как строки, числа, объекты и так далее. Массивы могут расти динамически. Проблемой массивов является низкая скорость работы и большое потребление памяти. Это отрицательно сказывается на производительности приложений с большими объемами вычислений и имеющими дело с большим количеством чисел. Появление буферных массивов решает эти проблемы.

Буферные массивы – это коллекции 8-битовых блоков в памяти. Каждый блок является элементом буферного массива. Размер буферного массива определяется при его создании и не может увеличиваться динамически. Буферные массивы могут хранить только числа. В момент создания буферного массива все его блоки инициализируются числом 0.

Объект буферного массива создается с помощью конструктора `ArrayBuffer`.

```
let buffer = new ArrayBuffer(80); // Размер 80 байт
```

Чтение и запись значений буферных массивов осуществляется с помощью объекта `DataView`. Не обязательно использовать для хранения чисел именно 8 бит. Точно так же можно использовать 16, 32 и 64 бита. Следующий пример демонстрирует, как создать объект `DataView` для чтения и записи данных из/в объект `ArrayBuffer`:

```
let buffer = new ArrayBuffer(80);
let view = new DataView(buffer);
view.setInt32(8,22,false);
var number = view.getInt32(8,false);
console.log(number); // Выведет "22"
```

В примере был создан объект `DataView` с помощью конструктора `DataView`. Объект `DataView` предоставляет несколько методов для чтения и записи чисел в буферные массивы. Мы воспользовались методом `setInt32()`, который использует 32 бита для хранения передаваемого числа.

Все методы объекта `DataView`, выполняющие запись данных в буферный массив, принимают три аргумента. Первый аргумент определяет смещение, то есть, количество байтов, которое следует отступить от начала массива перед записью числа. Второй аргумент представляет записываемое число. И третий аргумент (логического типа) определяет порядок записи байтов числа, например, если указать значение `false`, старшие байты будут записаны первыми (прямой порядок записи – big-endian).

Аналогично, все методы объекта `DataView`, выполняющие чтение данных из буферных массивов, принимают два аргумента. Первый определяет смещение, а второй – порядок следования байтов.

Ниже перечислены функции для сохранения чисел с помощью объекта `DataView`:

- **setInt8**: Использует 8 бит для хранения числа. Принимает целое число со знаком.
- **setUint8**: Использует 8 бит для хранения числа. Принимает целое число без знака.
- **setInt16**: Использует 16 бит для хранения числа. Принимает целое число со знаком.
- **setUint16**: Использует 16 бит для хранения числа. Принимает целое число без знака.
- **setInt32**: Использует 32 бита для хранения числа. Принимает целое число со знаком.
- **setUint32**: Использует 32 бита для хранения числа. Принимает целое число без знака.
- **setFloat32**: Использует 32 бита для хранения числа. Принимает вещественное число со знаком.
- **setFloat64**: Использует 64 бита для хранения числа. Принимает вещественное число со знаком.

Ниже перечислены функции для чтения чисел с помощью объекта `DataView`:

- **getInt8**: Читает 8 бит. Возвращает целое число со знаком.
- **getUint8**: Читает 8 бит. Возвращает целое число без знака.
- **getInt16**: Читает 16 бит. Возвращает целое число со знаком.
- **getUint16**: Читает 16 бит. Возвращает целое число без знака.
- **getInt32**: Читает 32 бита. Возвращает целое число со знаком.
- **getUint32**: Читает 32 бита. Возвращает целое число без знака.
- **getFloat32**: Читает 32 бита. Возвращает вещественное число со знаком.

- **getFloat64**: Читает 64 бита. Возвращает вещественное число со знаком.

Типизированные массивы

Мы рассмотрели чтение и запись чисел в буферные массивы. Но это очень громоздкий метод, так как каждый раз должна вызываться одна из функций. Типизированные массивы позволяют работать с буферными массивами как с обычными массивами.

Типизированный массив действует как обертка вокруг буферного массива и обрабатывает данные, как последовательности *n*-битных чисел. Значение *n* зависит от способа создания типизированного массива.

Следующий пример демонстрирует создание буферного массива, а также чтение и запись данных с помощью типизированного массива:

```
var buffer = new ArrayBuffer(80);
var typed_array = new Float64Array(buffer);
typed_array[4] = 11;
console.log(typed_array.length);
console.log(typed_array[4]);
```

Результат выполнения:

```
10
11
```

Здесь с помощью конструктора `Float64Array` был создан типизированный массив, интерпретирующий буферный массив как последовательность 64-битных вещественных чисел. Размер буферного массива равен 640 битам, поэтому в нем можно хранить не более десяти 64-битных чисел.

Существуют также другие виды типизированных массивов для доступа к буферным массивам как к последовательности чисел разной длины в битах. Они перечислены в следующем списке:

- **Int8Array**: для 8-битных целых чисел со знаком;
- **Uint8Array**: для 8-битных целых чисел без знака;
- **Int16Array**: для 16-битных целых чисел со знаком;
- **Uint16Array**: для 16-битных целых чисел без знака;
- **Int32Array**: для 32-битных целых чисел со знаком;
- **Uint32Array**: для 32-битных целых чисел без знака;
- **Float32Array**: для 32-битных вещественных чисел со знаком;
- **Float64Array**: для 64-битных вещественных чисел со знаком.

Типизированные массивы поддерживают все методы, доступные для обычных массивов JavaScript. Они также реализуют протокол итераций и, следовательно, могут использоваться в роли итерируемых объектов.

Объект Set

Объект **Set** является коллекцией *уникальных* значений любого типа. Значения в объекте `Set` хранятся в порядке их добавления. Объект `Set` создается с помощью конструктора `Set`. Следующий пример демонстрирует создание объекта Set:

```
let set1 = new Set();
let set2 = new Set("Hello!!!");
```

В примере переменная `set1` содержит пустой объект `Set`. В то время как переменная `set2`, созданная из итерируемого объекта, непустой строки символов, содержит непустой объект `Set`.

Следующий пример демонстрирует различные операции, доступные в объекте `Set`:

```
let set = new Set("Hello!!!");
set.add(12); // добавит в множество число 12
console.log(set.has("!")); // проверка существования значения
console.log(set.size);
set.delete(12); // удалит элемент с числом 12
console.log(...set);
set.clear(); // удалит все элементы
```

Результат выполнения:

```
true
6
H e l o !
```

В примере мы добавили девять элементов в объект `set`, но размер его равен шести, потому что объект `Set` автоматически удаляет повторяющиеся значения. Символы `l` и `!` повторяются несколько раз.

Объект `Set` также реализует протокол итераций, поэтому он может использоваться как итерируемый объект.

Объекты `Set` используются для хранения коллекций значений, если сценарию достаточно операции, позволяющей определить присутствие значения в коллекции, и не требуется извлекать сами значения. Например, объект `Set` можно использовать вместо массива, если

сценарий вызывает только метод `IndexOf()` массива для проверки существования значений.

Объект WeakSet

Ниже перечислены различия между объектами `Set` и `WeakSet`:

- объекты `Set` могут хранить значения элементарных типов и ссылки на объекты, а объект `WeakSet` может хранить только ссылки на объекты;
- одна из важных особенностей объекта `WeakSet`: если нет другой ссылки на объект, хранящийся в `WeakSet`, этот объект

```
map.set(o, "A"); // добавит пару ключ/значение
map.set("2", 9);
console.log(map.has("2")); // проверит наличие ключа
console.log(map.get(o)); // вернет значение, связанное с ключом
console.log(...map);
map.delete("2"); // удалит пару ключ/значение
map.clear(); // удалит все
// создать объект Map из итерируемого объекта
let map_1 = new Map([[1, 2], [4, 5]]);
console.log(map_1.size); // число ключей
```

Результат выполнения:

```
true
A
[object Object],A 2,9
2
```

При создании объекта `Map` из итерируемого объекта, этот объект должен возвращать массивы с 2 элементами, в которых индекс 0 соответствует ключу, а индекс 1 – значению.

Если попытаться добавить пару с ключом, который уже существует, новое значение затрет значение в существующей паре. Объекты `Map` реализуют протокол итераций и, следовательно, могут использоваться в роли итерируемых объектов. При выполнении итераций с объектами `Map`, они возвращают массивы с парами ключ/значение, как это показано в предыдущем примере.

Объект WeakMap

Ниже перечислены различия между объектами `Map` и `WeakMap`:

- ключи объекта `Map` могут быть значениями элементарных типов или ссылками на объекты, а ключи объекта `WeakMap` могут быть только ссылками на объекты;

```
let weakmap = new WeakMap();

(function(){
  let o = {n: 1};
  weakmap.set(o, "A");
})()

// здесь ключ 'o' будет уничтожен сборщиком мусора
let s = {m: 1};
weakmap.set(s, "B");
console.log(weakmap.get(s));
console.log(...weakmap); // Вызовет исключение
weakmap.delete(s);
weakmap.clear(); // Исключение, нет такой функции
let weakmap_1 = new WeakMap([[{}, 2], [{}, 5]]); // работает
console.log(weakmap_1.size); // Вернет undefined
```

Объект Object

Спецификация ES6 стандартизирует свойство `__proto__` объектов и добавляет новые свойства в глобальный объект `Object`.

Свойство __proto__

Каждый объект в JavaScript имеет внутреннее свойство `[[prototype]]` со ссылкой на прототип объекта, то есть, на наследуемый объект. Прочитать это можно с помощью метода `Object.getPrototypeOf()`, а, чтобы создать новый объект с заданным прототипом, следует использовать метод `Object.create()`. Свойство `[[prototype]]` объекта нельзя прочитать или изменить непосредственно.

Наследование в JavaScript выглядит громоздким, из-за особенностей свойства `[[prototype]]`, поэтому некоторые браузеры добавляют в объекты специальное свойство `__proto__`, обеспечивающее доступ к внутреннему свойству `[[prototype]]` и упрощающее работу с прототипами. Свойство `__proto__` не было стандартизировано в спецификации ES5, но из-за его популярности это было сделано в спецификации ES6.

Следующий пример демонстрирует использование свойства:

```
// В ES5
var x = {x: 12};
var y = Object.create(x, {y: {value: 13}});
console.log(y.x); // Выведет "12"
console.log(y.y); // Выведет "13"
```

```
// В ES6
let a = {a: 12, __proto__: {b: 13}};
console.log(a.a); // Выведет "12"
console.log(a.b); // Выведет "13"
```

Метод Object.is(value1, value2)

Метод `Object.is()` проверяет равенство двух значений. Он похож на оператор `===`, но имеет некоторые особенности, как показано в следующем примере:

```
console.log(Object.is(0, -0));
console.log(0 === -0);
console.log(Object.is(NaN, 0/0));
console.log(NaN === 0/0);
console.log(Object.is(NaN, NaN));
console.log(NaN === NaN);
```

Результат выполнения:

```
false
true
true
false
true
false
```

Метод Object.setPrototypeOf(object, prototype)

Метод `Object.setPrototypeOf()` — это всего лишь еще один способ присваивания значения свойству `[[prototype]]`. Следующий пример демонстрирует, как им пользоваться:

```
let x = {x: 12};
let y = {y: 13};
Object.setPrototypeOf(y, x)
console.log(y.x); // Выведет "12"
console.log(y.y); // Выведет "13"
```

Метод Object.assign(targetObj, sourceObjs...)

Метод `Object.assign()` копирует значения всех перечисляемых собственных свойств из одного или нескольких исходных объектов в целевой объект. Этот метод возвращает `targetObj`.

Следующий пример демонстрирует использование метода:

```
let x = {x: 12};
let y = {y: 13, __proto__: x};
let z = {z: 14, get b() {return 2;}, q: {}};
Object.defineProperty(z, "z", {enumerable: false});
let m = {};
Object.assign(m, y, z);
console.log(m.y);
console.log(m.z);
console.log(m.b);
console.log(m.x);
console.log(m.q == z.q);
```

Результат выполнения:

```
13
undefined
2
undefined
true
```

Ниже перечислены важные особенности, которые необходимо учитывать при использовании метода `Object.assign()`:

- вызывает методы чтения источников и методы записи приемника;
- просто присваивает значения свойств источника новым или существующим свойствам приемника;
- не копирует свойства `[[prototype]]` источников;
- имена свойств в JavaScript могут быть строками или символами, метод `Object.assign()` копирует и те и другие;
- определения свойств не копируются из источников, следовательно, для копирования определений нужно использовать методы `Object.getOwnPropertyDescriptor()` и `Object.defineProperty()`;
- игнорирует при копировании ключи со значениями `null` и `undefined`.

Итоги

В этой главе мы познакомились с новыми функциями, добавленными спецификацией ES6 для работы с числами, строками, массивами и объектами. Мы также узнали, что простые массивы часто являются причиной низкой производительности приложений, насыщенных

математическими расчетами, и как эту проблему можно решить с помощью буферных массивов. Мы также познакомились с новыми коллекциями, предоставляемыми спецификацией ES6.

В следующей главе мы поближе познакомимся с символами и протоколом итераций, а также с ключевым словом `yield` и генераторами.

ГЛАВА 3. Использование итераторов

Спецификация ES6 вводит новые объектные интерфейсы и способы выполнения итераций. Добавление новых итерационных протоколов открывает новый мир алгоритмов и возможностей для JavaScript. Мы начнем эту главу со знакомства с символами и различными свойствами объекта `Symbol`. Вы также узнаете, как создаются кадры в стеке вызовов для вложенных вызовов функций, о последствиях их создания, об оптимизации производительности и об оптимизации использования памяти.

Хотя символы является отдельной темой, не связанной напрямую с итераторами, мы рассмотрим их в этой главе, потому что они необходимы для реализации протокола итераций.

В этой главе мы рассмотрим:

- использование символов как ключей свойств объектов;
- реализацию протоколов итераций в объектах;
- создание и использование объектов-генераторов;
- использование циклов `for...of` для итераций;
- оптимизацию хвостовых вызовов.

Символы в спецификации ES6

Символы в ES6 являются новым элементарным типом. Символ является уникальным и неизменным значением. Следующий пример демонстрирует создание символа:

```
var s = Symbol();
```

Символы не имеют литеральной формы, следовательно, для создания символов необходимо использовать функцию `Symbol()`. Функция `Symbol()` возвращает уникальный символ при каждом своем вызове.

Функция `Symbol()` принимает необязательный строковый параметр с описанием символа. Описания символов можно использовать

при отладке, но они не имеют доступа к самому символу. Два символа с тем же описанием не равны друг другу. Вот пример, демонстрирующий это:

```
let s1 = window.Symbol("My Symbol");
let s2 = window.Symbol("My Symbol");
console.log(s1 === s2); // Выведет "false"
```

Из предыдущего примера, мы можем также сказать, что символ имеет значение, *похожее на строку*, которое не может вступить в конфликт с любым другим значением.

Оператор typeof

Результатом применения оператора `typeof` к переменной, содержащий символ, является строка `symbol`. Пример, демонстрирующий это:

```
var s = Symbol();
console.log(typeof s); // Выведет "symbol"
```

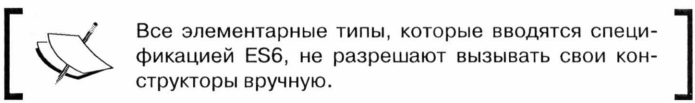
Оператор `typeof` – это единственный способ определить, что переменная содержит символ.

Оператор new

Оператор `new` нельзя применить к функции `Symbol()`. Функция `Symbol()` сама определяет, используется ли она в качестве конструктора и, если это так, вызывает исключение. Например:

```
try
{
  let s = new Symbol(); // Исключение "TypeError"
}
catch(e)
{
  console.log(e.message); // Выведет "Symbol is not a constructor"
}
```

Реализации JavaScript могут внутренне использовать функцию `Symbol()` в качестве конструктора для обертывания символа объектом. Таким образом, переменная `s` будет равна `Object(s)`.



Все элементарные типы, которые вводятся спецификацией ES6, не разрешают вызывать свои конструкторы вручную.

Использование символов как ключей свойств

В спецификации ES5, и более ранних, ключи свойств объекта JavaScript должны были быть строками. Но в спецификации ES6 ключи свойств объектов JavaScript могут быть как строками, так и символами. Следующий пример демонстрирует использование символа в качестве ключа свойства объекта:

```
let obj = null;
let s1 = null;
(function(){
  let s2 = Symbol();
  s1 = s2;
  obj = {[s2]: "mySymbol"}
  console.log(obj[s2]);
  console.log(obj[s2] == obj[s1]);
})();
console.log(obj[s1]);
```

Результат выполнения:

```
mySymbol
```

`getOwnPropertySymbols()`, возвращающий массив символьных свойств объекта. Следующий пример демонстрирует его использование:

```
let obj = {a: 12};
let s1 = Symbol("mySymbol");
let s2 = Symbol("mySymbol");
Object.defineProperty(obj, s1, {
  enumerable: false
});
obj[s2] = "";
console.log(Object.getOwnPropertySymbols(obj));
```

Результат выполнения:

```
Symbol(mySymbol),Symbol(mySymbol)
```

Из предыдущего примера видно, что метод `Object.getOwnPropertySymbols()` может извлекать и неперечисляемые символьные свойства.

Оператор `in` обнаруживает символьные свойства объекта, в то время как цикл `for...in` и метод `Object.getOwnPropertyNames()` – нет, для поддержания обратной совместимости.

Метод Symbol.for(string)

Объект `Symbol` ведет регистрацию пар ключ/значение, где ключом является описание символа, а значением – сам символ. Всякий раз, когда вызовом `Symbol.for()` создается символ, он добавляется в реестр и метод возвращает символ. Если попытаться создать символ с описанием, которое уже существует, будет извлечен существующий символ.

Преимуществом использования `Symbol.for()` вместо `Symbol()` для создания символов является отсутствие необходимости беспокоиться о придании символам глобальной области видимости, потому `Symbol.for()` всегда создает символы, доступные глобально. Следующий пример демонстрирует это:

```
let obj = {};
(function(){
  let s1 = Symbol("name»);
  obj[s1] = "Eden";
})();
// obj[s1] недоступен здесь
(function(){
```

```
  let s2 = Symbol.for("age");
  obj[s2] = 27;
})();
console.log(obj[Symbol.for("age")]); // Выведет "27"
```

Встроенные символы

Спецификация ES6 не только позволяет создавать собственные символы, но и предоставляет набор встроенных символов, называемых **популярными** (well-known) символами. Вот список свойств, ссылающихся на некоторые самые важные встроенные символы:

- `Symbol.iterator`
- `Symbol.match`
- `Symbol.search`
- `Symbol.replace`
- `Symbol.split`
- `Symbol.hasInstance`
- `Symbol.species`
- `Symbol.unscopables`
- `Symbol.isContcatSpreadable`
- `Symbol.toPrimitive`
- `Symbol.toStringTag`

Вы еще столкнетесь с этими символами в других главах этой книги.

При ссылках на популярные символы в тексте обычно используется префикс `@@`. Например, символ `Symbol.iterator` обозначается, как `@@iterator`. Это сделано, для упрощения обозначения популярных символов в тексте.

Протоколы итераций

Протокол итераций представляет набор правил, которым должен следовать объект для реализации интерфейса, используемого циклами или конструкциями итераций по набору значений объекта.

Спецификация ES6 вводит два новых протокола итераций: **итерационный протокол** (iterable protocol) и **протокол итератора** (iterator protocol).

Протокол итератора

Любой объект, реализующий протокол итератора, называется итератором. В соответствии с протоколом итератора, объект должен реали-

зовать метод `next()`, возвращающий следующий элемент в последовательности групп элементов.

Следующий пример демонстрирует это:

```
let obj = {
  array: [1, 2, 3, 4, 5],
  nextIndex: 0,
  next: function(){
    return this.nextIndex < this.array.length ?
    {value: this.array[this.nextIndex++], done: false} :
    {done: true};
  }
};
console.log(obj.next().value);
console.log(obj.next().value);
console.log(obj.next().value);
console.log(obj.next().value);
console.log(obj.next().value);
console.log(obj.next().done);
```

Результат выполнения примера:

```
1
2
3
4
5
true
```

Каждый раз, когда вызывается метод `next()`, он возвращает объект с двумя свойствами: `value` и `done`. Давайте посмотрим, что представляют собой эти два свойства:

- свойство `done` возвращает `true`, если итератор закончил обход коллекции значений, иначе оно возвращает `false`;
- свойство `value` хранит значение текущего элемента коллекции, оно опускается, когда свойство `done` возвращает `true`.

Итерационный протокол

Любой объект, реализующий итерационный протокол, называется итерируемым. Согласно итерационному протоколу, объект должен предоставить метод `@@iterator`, то есть, иметь свойство с символьным ключом `Symbol.iterator`. Метод `@@iterator` должен возвращать объект-итератор.

Следующий пример демонстрирует это:

```
let obj = {
array: [1, 2, 3, 4, 5],
nextIndex: 0,
[Symbol.iterator]: function(){
  return {
    array: this.array,
    nextIndex: this.nextIndex,
    next: function(){
      return this.nextIndex < this.array.length ?
      {value: this.array[this.nextIndex++], done: false} :
        {done: true};
      }
    }
  }
};
let iterable = obj[Symbol.iterator]()
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().done);
```

Результат выполнения примера:

```
1
2
3
4
5
true
```

Генераторы

Функция-генератор похожа на обычную функцию, но возвращает не одно значение, а несколько значений по одному. При вызове функция-генератор выполняет код в своем теле не весь сразу, а возвращает новый экземпляр объекта-генератора (который реализует оба протокола, итерационный и итератора).

Каждый объект-генератор содержит новый исполняемый контекст функции-генератора. Когда вызывается метод `next()` объекта-генератора, он выполняет тело функции-генератора до ключевого слова `yield`. Это ключевое слово возвращает значение и приостанавливает функцию. Когда метод `next()` вызывается снова, функция возобновляет выполнение и возвращает следующее значение. Свойство `done` получает значение `true`, когда функция-генератор вернет все значения.

Функция-генератор определяется с помощью выражения `function*`. Следующий пример демонстрирует его использование:

```
function* generator_function()
{
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
}
let generator = generator_function();
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().done);
generator = generator_function();
let iterable = generator[Symbol.iterator]();
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().value);
console.log(iterable.next().done);
```

Результат выполнения примера:

```
1
2
3
4
5
true
1
2
3
4
5
true
```

За ключевым словом `yield` следует выражение. Значение этого выражения возвращается функцией-генератором с помощью итерационного протокола. Если опустить выражение, функция вернет `undefined`. Значение выражения называется значением для выдачи.

Метод `next()` принимает один необязательный аргумент. Этот аргумент становится значением, возвращаемым оператором `yield`,

который приостановил функцию-генератор. Следующий пример демонстрирует это:

```
function* generator_function()
{
  var a = yield 12;
  var b = yield a + 1;
  var c = yield b + 2;
  yield c + 3;
}
var generator = generator_function();
console.log(generator.next().value);
console.log(generator.next(5).value);
console.log(generator.next(11).value);
console.log(generator.next(78).value);
console.log(generator.next().done);
```

Результат выполнения примера:

```
12
6
13
81
true
```

Метод return(value)

Выполнение функции-генератора можно завершить в любой момент, до того, как она вернет все значения, вызвав метод `return()` объекта-генератора. Метод `return()` принимает необязательный аргумент с последним возвращаемым значением.

Следующий пример демонстрирует метод:

```
function* generator_function()
{
  yield 1;
  yield 2;
  yield 3;
}
var generator = generator_function();
console.log(generator.next().value);
console.log(generator.return(22).value);
console.log(generator.next().done);
```

Результат выполнения примера:

```
1
22
true
```

Метод throw(exception)

Внутри функции-генератора можно вручную вызвать исключение с помощью метода `throw()` объекта-генератора. Методу `throw()`, при этом, следует передать исключение, которое требуется возбудить. Следующий пример демонстрирует это:

```
function* generator_function()
{
  try
  {
    yield 1;
  }
  catch(e)
  {
    console.log("1st Exception");
  }
  try
  {
    yield 2;
  }
  catch(e)
  {
    console.log("2nd Exception");
  }
}
var generator = generator_function();
console.log(generator.next().value);
console.log(generator.throw("exception string").value);
console.log(generator.throw("exception string").done);
```

Результат выполнения примера:

```
1
1st Exception
2
2nd Exception
true
```

В предыдущем примере видно, что исключение возбуждается в месте, где функция была приостановлена в последний раз. После обработки исключения, метод `throw()` продолжил выполнение и вернул следующее значение для выдачи.

Ключевое слово yield*

Ключевое слово `yield*` внутри функции-генератора принимает итерируемый объект и выполняет итерации по нему, возвращая полученные из него значения. Следующий пример демонстрирует это:

```
function* generator_function_1()
{
  yield 2;
  yield 3;
}
function* generator_function_2()
{
  yield 1;
  yield* generator_function_1();
  yield* [4, 5];
}
var generator = generator_function_2();
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().done);
```

Результат выполнения примера:

```
1
2
3
4
5
true
```

Цикл for...of

До сих пор мы выполняли перебор значений итерируемого объекта с помощью метода `next()`, что достаточно трудоемко. В спецификации ES6 введен цикл `for...of` для облегчения этой задачи.

Цикл `for...of` был добавлен для обхода значений итерируемого объекта. Следующий пример демонстрирует его использование:

```
function* generator_function()
{
  yield 1;
  yield 2;
  yield 3;
  yield 4;
  yield 5;
}
let arr = [1, 2, 3];
for(let value of generator_function())
{
  console.log(value);
}
```

```
for(let value of arr)
{
  console.log(value);
}
```

Результат выполнения примера:

```
1
2
3
4
5
1
2
3
```

Оптимизация хвостового вызова

Всякий раз, когда вызывается функция, в стеке вызовов создается новый кадр для хранения локальных переменных функции.

Когда функция вызывается из другой функции, создается еще один кадр в стеке вызовов для внутренней функции. Проблема состоит в том, что кадр стека внутренней функции занимает некоторое количество дополнительной памяти, так как хранит адрес возврата для возобновления выполнения после завершения внутренней функции. Переключение и создание кадров стека вызовов также занимает дополнительное процессорное время. Эта проблема не оказывает заметного влияния, когда имеется пара или сотня уровней вложенности вызовов, но когда уровень вложенности достигает тысячи и более, реализации JavaScript возбуждают исключение `RangeError: Maximum call stack size exceeded`. Возможно, вам уже приходилось сталкиваться с ошибкой `RangeError` при создании рекурсивных функций.

Хвостовым вызовом (tail call) является вызов функции, осуществляемый в самом конце функции оператором `return`. Если хвостовой вызов ведет в ту же самую функцию снова и снова, он называется **хвостовой рекурсией** (tail-recursion), которая является частным случаем рекурсии. Особенность хвостового вызова в том, что есть возможность избежать дополнительных расходов процессорного времени и памяти за счет повторного использования существующего кадра стека функции вместо создания нового. Повторное использование кадра стека при хвостовом вызове называется **оптимизацией хвостового вызова**.

Спецификация ES6 добавляет поддержку оптимизации хвостовых вызовов, если сценарий использует режим `"use strict"`. Давайте рассмотрим пример хвостового вызова:

```
"use strict";
function _add(x, y)
{
  return x + y;
}
function add1(x, y)
{
  x = parseInt(x);
  y = parseInt(y);
  // хвостовой вызов
  return _add(x, y);
}
function add2(x, y) {
  x = parseInt(x);
  y = parseInt(y);
  // этот вызов не является хвостовым
  return 0 + _add(x, y);
}
console.log(add1(1, '1')); //2
console.log(add2(1, '2')); //3
```

Здесь вызов функции `_add()` из функции `add1()` является хвостовым вызовом, так как это последнее действие в функции `add1()`. Но вызов функции `_add()` из функции `add2()` не является хвостовым вызовом, так как это не последнее действие в функции `add2()`, здесь последним действием является добавление `0` к результату функции `_add()`.

Вызов функции `_add()` из функции `add1()` не создает нового кадра в стеке вызовов. Вместо этого, он повторно использует стек выполнения функции `add1()`, другими словами, происходит оптимизация хвостового вызова.

Преобразование неконцевых вызовов в концевые вызовы

Так как хвостовые вызовы оптимизируются, используйте хвостовые вызовы вместо обычных везде, где это только возможно. Вы можете оптимизировать свой код путем преобразования обычных вызовов в хвостовые. Рассмотрим пример преобразования обычных вызовов в хвостовые:

```
"use strict";
function _add(x, y)
{
  return x + y;
}
function add(x, y)
{
  x = parseInt(x);
  y = parseInt(y);
  var result = _add(x, y);
  return result;
}
console.log(add(1, '1'));
```

Здесь вызов функции `_add()` не является хвостовым, поэтому было создано два кадра в стеке вызовов. Мы можем преобразовать его в хвостовой, как показано ниже:

```
function add(x, y)
{
  x = parseInt(x);
  y = parseInt(y);
  return _add(x, y);
}
```

Здесь мы отказались от переменной `result` и совместили вызов функции с оператором `return` в одной строке. Существует много других стратегий преобразования обычных вызовов в хвостовые.

Итоги

В этой главе, мы узнали новый способ создания ключей свойств объектов с помощью символов. Познакомились с протоколами итераторов и итерационными протоколами, и узнали, как реализовать эти протоколы для пользовательских объектов. Далее мы посмотрели,

Асинхронный код, основанный на событиях

Для использования асинхронных программных интерфейсов JavaScript, основанных на событиях, необходимо зарегистрировать обработчики событий успешного и неудачного завершения операции, которые будут выполняться, в зависимости от успеха или неудачи операции.

Например, перед выполнением AJAX-запроса нужно зарегистрировать обработчики событий, один из которых будет выполнен, в зависимости от успеха или неудачи AJAX-запроса. Рассмотрим следующий фрагмент кода, который выполняет AJAX-запрос и выводит полученную информацию:

```
function displayName(json)
{
  try
  {
    // обычно вывод выполняется с использованием DOM
    console.log(json.Name);
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
}
function displayProfession(json)
{
  try
  {
    console.log(json.Profession);
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
}
function displayAge(json)
{
  try
  {
    console.log(json.Age);
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
```

```
}
function displayData(data)
{
  try
  {
    var json = JSON.parse(data);
    displayName(json);
    displayProfession(json);
    displayAge(json);
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
}
var request = new XMLHttpRequest();
var url = "data.json";
request.open("GET", url);
request.addEventListener("load", function(){
  if(request.status === 200)
  {
    displayData(request.responseText);
  }
  else
  {
    console.log("Server Error: " + request.status);
  }
}, false);
request.addEventListener("error", function(){
  console.log("Cannot Make AJAX Request");
}, false);
request.send();
```

Здесь предполагается, что файл `data.json` содержит следующий текст:

```
{
  "Name": "Eden",
  "Profession": "Developer",
  "Age": "25"
}
```

Метод `send()` объекта `XMLHttpRequest`, выполняемый асинхронно, извлекает файл `data.json` и вызывает обработчик события `load` или `error` в зависимости от успеха запроса.

Работа такого AJAX-запроса, основанного на событиях, не вызывает никаких нареканий, проблема только в коде, который пришлось написать. Ниже перечисляются сложности, с которыми пришлось столкнуться при написании предыдущего примера:

- Потребовалось добавить обработчик исключений в каждый блок кода, выполняющийся асинхронно. Нельзя просто поместить весь код в один оператор `try...catch`. Это усложнит обработку исключений.
- Код сложно читать, так как трудно отследить поток выполнения во вложенных вызовах функций.
- Если в другом фрагменте программы нужно будет узнать о состоянии асинхронной операции (завершилась, приостановлена или выполняется), для этого придется использовать дополнительные переменные. То есть, можно констатировать трудность определения состояния асинхронной операции.

Этот код может получиться еще более сложным и трудночитаемым, если будет содержать несколько AJAX-запросов или других асинхронных операций. Например, после отображения данных, может понадобиться попросить пользователя проверить, верны ли выведенные данные или нет, а затем отправить логическое значение обратно на сервер. Следующий пример демонстрирует это:

```
function verify()
{
  try
  {
    var result = confirm("Is the data correct?");
    if (result == true)
    {
      // выполнить AJAX-запрос к серверу
    }
    else
    {
      // выполнить AJAX-запрос к серверу
    }
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
}
function displayData(data)
{
  try
  {
    var json = JSON.parse(data);
    displayName(json);
    displayProfession(json);
    displayAge(json);
```

```
      verify();
    }
    catch(e)
    {
      console.log("Exception: " + e.message);
    }
  }
```

Асинхронный код, основанный на обратных вызовах

При использовании асинхронных программных интерфейсов JavaScript, основанных на обратных вызовах, требуется передать пару функций обратного вызова, одна из которых будет вызвана, в зависимости успеха или неудачи операции.

Например, выполняя AJAX-запрос с помощью JQuery, необходимо передать две функции обратного вызова, одна из которых будет вызвана в случае успеха, а другая – в случае неудачи. Рассмотрим следующий фрагмент, который выполняет AJAX-запрос с помощью JQuery и выводит полученную информацию:

```
function displayName(json)
{
  try
  {
    console.log(json.Name);
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
}
function displayProfession(json)
{
  try
  {
    console.log(json.Profession);
  }
  catch(e)
  {
    console.log("Exception: " + e.message);
  }
}
function displayAge(json)
{
  try
  {
```

```
        console.log(json.Age);
      }
      catch(e)
      {
        console.log("Exception: " + e.message);
      }
    }
    function displayData(data)
    {
      try
      {
        var json = JSON.parse(data);
        displayName(json);
        displayProfession(json);
        displayAge(json);
      }
      catch(e)
      {
        console.log("Exception: " + e.message);
      }
    }
     $.ajax({url: "data.json", success: function(result, status,
responseObject){
      displayData(responseObject.responseText);
    }, error: function(xhr,status,error){
     console.log("Cannot Make AJAX Request. Error is: " + error);
    }});
```

И здесь работа JQuery не вызывает никаких нареканий, проблема опять только в коде, который пришлось написать. Ниже перечислены сложности, с которыми пришлось столкнулись при создании предыдущего примера:

- Трудно перехватывать исключения, так как необходимо использовать несколько операторов `try` и `catch`.
- Сложно читать код, так как трудно уследить за потоком выполнения во вложенных вызовах функций.
- Трудно отслеживать состояние асинхронной операции.

И этот код станет гораздо сложнее при включении в него нескольких AJAX-запросов JQuery или других асинхронных операций.

Объекты Promise в помощь

Спецификация ES6 вводит новый встроенный объект Promise, реализующий асинхронные операции.

Этот новый объект решает проблемы, возникающие при применении программных интерфейсов событий и обратных вызовов. Он делает асинхронный код похожим на синхронный.

Объект `Promise` представляет асинхронную операцию. Существующие асинхронные программные интерфейсы JavaScript обычно обертываются объектами `Promise`, а новые, современные программные интерфейсы JavaScript реализуются с помощью самих объектов `Promise`.

Объект `Promise` является новинкой для JavaScript, но он давно существует во многих других языках программирования. Языки программирования, такие как C# 5, C++ 11, Swift, Scala и многие другие, поддерживают объект `Promise`.

Спецификация ES6 вводит программный интерфейс Promise API, с помощью которого можно создавать объекты `Promise` и использовать их. Давайте рассмотрим программный интерфейс Promise API, определяемый спецификацией ES6.

Конструктор Promise

Конструктор `Promise` служит для создания новых экземпляров `Promise`. Объект `Promise` представляет асинхронную операцию.

Мы должны передать в конструктор `Promise` свою функцию обратного вызова, выполняющую асинхронную операцию. Эта функция называется **исполнителем** (executor). Исполнитель должен принимать два параметра – функции обратного вызова `resolve` и `reject`. Функция `resolve` будет вызвана в случае успеха асинхронной операции, а функция `reject` – в случае неудачи. Если асинхронная операция прошла успешно и имеет результат, он будет передан в функцию обратного вызова `resolve`. Если асинхронная операция завершилась неудачей, в функцию обратного вызова `reject` будет передано описание причины.

Следующий пример демонстрирует создание объекта `Promise` и обертывание им AJAX-запроса:

```
var promise = new Promise(function(resolve, reject){
  var request = new XMLHttpRequest();
  var url = "data.json";
  request.open("GET", url);
  request.addEventListener("load", function(){
    if(request.status === 200)
    {
      resolve(request.responseText);
    }
```

```
        else
        {
          reject("Server Error: " + request.status);
        }
      }, false);
      request.addEventListener("error", function(){
        reject("Cannot Make AJAX Request");
      }, false);
      request.send();
    })
```

Исполнитель выполняется синхронно, но он выполняет асинхронную операцию и, следовательно, может вернуть управление до завершения асинхронной операции.

Объект `Promise` всегда находится в одном из следующих состояний:

- **выполнено** (Fulfilled): если вызвана функция `resolve` с аргументом, не являющимся объектом `Promise`, или без аргументов, асинхронная операция считается выполненной;
- **отклонено** (Rejected): если вызвана функция `reject` или возникло исключение внутри исполнителя, асинхронная операция считается отклоненной;
- **ожидание** (Pending): если ни одна из функций – `resolve` и `reject` – еще не вызвана, асинхронная операция находится в режиме ожидания;
- **завершено** (Settled): асинхронная операция считается завершенной, если она выполнена или отклонена, и не находится в режиме ожидания.

После того, как асинхронная операция будет выполнена или отклонена, она уже не сможет сменить состояние. Попытка смены состояния не будет иметь никакого эффекта.

Если функция обратного вызова `resolve` будет вызвана с объектом `Promise` в качестве аргумента, асинхронная операция будет либо выполнена либо отклонена, в зависимости от того будет ли асинхронная операция, представленная переданным объектом `Promise`, выполнена или отклонена.

Результат асинхронной операции

В случае успеха асинхронная операция, представленная объектом `Promise`, возвращает результат.

Если аргумент, переданный функции `resolve`, содержит любое значение, не являющееся объектом `Promise`, его можно рассматривать как результат асинхронной операции.

Если вызвать `resolve` без аргумента, значением асинхронной операции станет `undefined`, а сама операция будет считается выполненной.

Чтобы увидеть, что происходит, при передаче функции `resolve` объекта `Promise`, рассмотрим следующий пример. Предположим, что есть объект `Promise` с именем *A*. При вызове функции `resolve` объекта *A*, ей был передан объект `Promise` с именем *B*, тогда объект *A* (т. е. асинхронная операция, которую он представляет) будет считаться выполненным, если объект *B* будет выполнен, и результат выполнения объекта *A*, будет совпадать с результатом выполнения объекта *B*.

Рассмотрим пример:

```
var a = new Promise(function(resolve, reject){
  var b = new Promise(function(res, rej){
    rej("Reason");
  });
  resolve(b);
});
var c = new Promise(function(resolve, reject){
  var d = new Promise(function(res, rej){
    res("Result");
  });
  resolve(d);
});
```

В предыдущем примере, объект *B* был отклонен, следовательно, объект *A* также будет отклонен. Причину отклонения обоих объектов будет описывать строка `"Reason"`. Точно так же, *C* будет считаться выполненным, потому что объект *D* выполнен. Результатом выполнения *C* и *D* будет строка `"Result"`.

Когда мы говорим, что "Объект Promise разрешается со значением, или разрешен со значением", это значит, что исполнитель Promise вызывает или вызвал функцию обратного вызова `resolve` с результатом.

Метод then(onFulfilled, onRejected)

Метод `then()` объекта `Promise` позволяет выполнить некоторые действия, после того, как асинхронная операция будет выполнена или отклонена. Эти действия могут быть другими событиями или обратны-

ми вызовами, основанными на асинхронных операциях.

Метод `then()` объекта `Promise` принимает два аргумента – функции обратного вызова `onFulfilled` и `onRejected`. Функция `onFulfilled` выполняется, если объект асинхронная операция была выполнена, а функция `onRejected` выполняется, если асинхронная операция была отклонена.

Кроме того, функция обратного вызова `onRejected` вызывается также, если возникло исключение в исполнителе. То есть, она ведет себя как обработчик исключений, перехватывая исключения.

Функция обратного вызова `onFulfilled` принимает параметр – результат выполнения асинхронной операции. Аналогично, функция обратного вызова `onRejected` принимает параметр, описывающий причину отказа.

Функции, передаваемые методу `then()`, выполняются асинхронно.

Следующий пример демонстрирует использование метода `then()`:

```
var promise = new Promise(function(resolve, reject){
  var request = new XMLHttpRequest();
  var url = "data.json";
  request.open("GET", url);
  request.addEventListener("load", function(){
    if(request.status === 200)
    {
      resolve(request.responseText);
    }
    else
    {
      reject("Server Error: " + request.status);
    }
  }, false);
  request.addEventListener("error", function(){
    reject("Cannot Make AJAX Request");
    }, false);
  request.send();
});
promise.then(function(value){
  value = JSON.parse(value);
  return value;
}, function(reason){
  console.log(reason);
});
```

Здесь, если AJAX-запрос выполнится успешно (то есть, асинхронная операция будет выполнена), функция `onFulfilled` получит текст ответа, преобразует JSON-строку в объект JavaScript и вернет его.

Многие программисты обычно удаляют переменную с объектом

Promise и записывают предыдущий пример так:

```
function ajax()
{
  return new Promise(function(resolve, reject){
    var request = new XMLHttpRequest();
    var url = "data.json";
    request.open("GET", url);
    request.addEventListener("load", function(){
      if(request.status === 200)
      {
        resolve(request.responseText);
      }
      else
      {
        reject("Server Error: " + request.status);
      }
    }, false);
    request.addEventListener("error", function(){
      reject("Cannot Make AJAX Request");
    }, false);
    request.send();
  });
}
ajax().then(function(value){
  value = JSON.parse(value);
  return value;
}, function(reason){
  console.log(reason);
});
```

Такой подход делает код еще более простым для чтения. Все новые программные интерфейсы JavaScript, реализованные с использованием объектов Promise, следуют этому шаблону.

Метод then() всегда возвращает новый объект Promise, которой разрешает возвращаемое значение функции обратного вызова. Ниже перечислены случаи, в которых метод then() возвращает новый объект Promise:

- Если вызванная функция onFulfilled не содержит оператора возврата, будет возвращен неявно созданный объект Promise, представляющий завершенную асинхронную операцию.
- Если вызванная функция onFulfilled возвращает пользовательский объект Promise, неявно создается и возвращается новый объект Promise. Новый объект Promise разрешает пользовательский объект Promise.

- Если вызванная функция `onFulfilled` возвращает нечто, отличное от объекта `Promise`, так же неявно создается и возвращается новый объект `Promise`. Новый объект `Promise` разрешает возвращаемое значение.
- Если вместо функции обратного вызова `onFulfilled` передано значение `null`, неявно создается функция обратного вызова и замещает значение `null`. Неявно созданная функция `onFulfilled` возвращает новый объект `Promise`. Результат выполнения нового объекта `Promise` совпадает с результатом выполнения родительского объекта `Promise`.
- Если вызванная функция `onRejected` не содержит оператора возврата, будет возвращен неявно созданный отклоненный объект `Promise`.
- Если вызванная функция обратного вызова `onRejected` возвращает пользовательский объект `Promise`, неявно создается и возвращается новый объект `Promise`. Новый объект `Promise` разрешает пользовательский объект `Promise`.
- Если вызванная функция `onRejected` возвращает нечто, отличное от объекта `Promise`, так же неявно создается и возвращается новый объект `Promise`. Новый объект `Promise` разрешает возвращаемое значение.
- Если вместо функции обратного вызова `onRejected` передано значение `null`, неявно создается функция обратного вызова и замещает значение `null`. Неявно созданная функция `onRejected` возвращает новый отклоненный объект. Причина отклонения нового объекта `Promise` совпадает с причиной отклонения родительского объекта `Promise`.

В предыдущем примере ничего не выводилось в консоль. Чтобы получить такой вывод, создадим цепь объектов `Promise`. Кроме того, в предыдущем примере, мы не обрабатывали исключения, которые могут возникнуть в функции `onFulfilled`. В следующем примере добавим вывод данных в консоль и обработку исключений, которые могли возникнуть во всех, связанных в цепь функциях `onFulfilled`:

```
function ajax()
{
  return new Promise(function(resolve, reject){
    var request = new XMLHttpRequest();
    var url = "data.json";
    request.open("GET", url);
    request.addEventListener("load", function(){
```

```
        if(request.status === 200)
        {
          resolve(request.responseText);
        }
        else
        {
          reject("Server Error: " + request.status);
        }
      }, false);
      request.addEventListener("error", function(){
        reject("Cannot Make AJAX Request");
      }, false);
      request.send();
    });
  }
  ajax().then(function(value){
    value = JSON.parse(value);
    return value;
  }).then(function(value){
    console.log(value.Name);
    return value;
  }).then(function(value){
    console.log(value.Profession);
    return value;
  }).then(function(value){
    console.log(value.Age);
    return value;
  }).then(null, function(reason){
    console.log(reason);
  });
```

В этом примере несколько объектов `Promise` связаны в цепь с помощью метода `then()` для разбора и вывода ответа, полученного исполнителем первого объекта `Promise`. Здесь последний метод `then()` используется в качестве обработчика исключений или ошибок для всех методов `onFulfilled` и исполнителей.

Следующая схема демонстрирует работу нескольких объектов `Promise`, связанных в цепь:

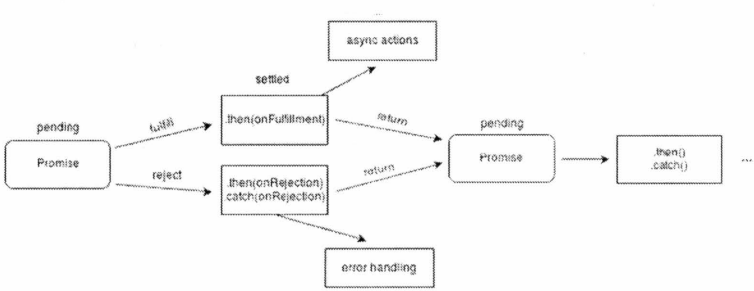


Рис. 4.1. Изображение предоставлено MDN

Давайте продолжим и добавим в цепь, основанную на событиях, асинхронную операцию, чтобы подтвердить правильность отображенных данных. Для этого дополним код следующим образом:

```
function ajax()
{
  return new Promise(function(resolve, reject){
    var request = new XMLHttpRequest();
    var url = "http://localhost:8888/data.json";
    request.open("GET", url);
    request.addEventListener("load", function(){
      if(request.status === 200)
      {
        resolve(request.responseText);
      }
      else
      {
        reject("Server Error: " + request.status);
      }
    }, false);
    request.addEventListener("error", function(){
      reject("Cannot Make AJAX Request");
    }, false);
    request.send();
  });
}
function verify(value)
{
  return new Promise(function(resolve, reject){
    if(value == true)
    {
      // выполнить AJAX-запрос и отправить данные на сервер
    }
    else
    {
      // выполнить AJAX-запрос и отправить данные на сервер
    }
  });
}
ajax().then(function(value){
  value = JSON.parse(value);
  return value;
}).then(function(value){
  console.log(value.Name);
  return value;
}).then(function(value){
  console.log(value.Profession);
  return value;
}).then(function(value){
  console.log(value.Age);
```

```
    return value;
  }).then(function(value){
    var result = confirm("Is the data correct?");
    return result;
  }).then(verify).then(null, function(reason){
    console.log(reason);
  });
```

Этот пример демонстрирует, как обертывание операции AJAX с помощью объектов `Promise` упрощает код. Теперь код стал более понятным.

Метод catch(onRejected)

Метод `catch()` объекта `Promise` применяется вместо метода `then()` там, где метод `then()` используется исключительно для обработки ошибок и исключений. В работе метода `catch()` нет никаких особенностей. Он просто делает код более удобным для чтения, поскольку слово «catch» точнее отражает назначение кода.

Метод `catch()` принимает один аргумент, функцию обратного вызова `onRejected`. Функция `onRejected` вызывается так же, как функция `onRejected` метода `then()`.

Метод `catch()` всегда возвращает объект `Promise`. Ниже перечисляются случаи, когда создается новый объект `Promise`, возвращаемый методом `catch()`:

- Если функция обратного вызова `onRejected` не содержит оператора возврата, будет возвращен неявно созданный новый выполненный объект `Promise`.
- Если вызванная функция `onRejected` возвращает пользовательский объект `Promise`, неявно создается и возвращается новый объект `Promise`. Новый объект `Promise` разрешает пользовательский объект `Promise`.
- Если вызванная функция `onRejected` возвращает нечто отличное от объекта `Promise`, так же неявно создается и возвращается новый объект. Новый объект `Promise` разрешает возвращаемое значение.
- Если вместо функции обратного вызова `onRejected` передано значение `null` или соответствующий аргумент вообще опущен, неявно создается функция обратного вызова и используется как аргумент. Неявно созданная функция `onRejected` возвращает новый отклоненный объект `Promise`. Причина

отклонения нового объекта `Promise` совпадает с причиной отклонения родительского объекта `Promise`.

- Если объект `Promise`, из которого вызван метод `catch()`, является выполненным, метод `catch()` просто возвращает новый выполненный объект `Promise`, игнорируя обратный вызов `onRejected`. Результат выполнения нового объекта `Promise` совпадает с результатом родительского объекта `Promise`.

Чтобы разобраться с методом `catch()`, рассмотрим пример:

```
promise.then(null, function(reason){
});
```

Этот вызов можно переписать с использованием метода `catch()`:

```
promise.catch(function(reason){
});
```

Эти два фрагмента работают совершенно одинаково.

Перепишем пример AJAX-вызова, заменив последний метод `then()` методом `catch()`:

```
function ajax()
{
  return new Promise(function(resolve, reject){
    var request = new XMLHttpRequest();
    var url = "data.json";
    request.open("GET", url);
    request.addEventListener("load", function(){
      if(request.status === 200)
      {
        resolve(request.responseText);
      }
      else
      {
        reject("Server Error: " + request.status);
      }
    }, false);
    request.addEventListener("error", function(){
      reject("Cannot Make AJAX Request");
    }, false);
    request.send();
  });
}
function verify(value)
{
  return new Promise(function(resolve, reject){
    if(value == true)
    {
```

```
        // выполнить AJAX-запрос и отправить данные на сервер
      }
      else
      {
        // выполнить AJAX-запрос и отправить данные на сервер
      }
    });
  }
  ajax().then(function(value){
    value = JSON.parse(value);
    return value;
  }).then(function(value){
    console.log(value.Name);
    return value;
  }).then(function(value){
    console.log(value.Profession);
    return value;
  }).then(function(value){
    console.log(value.Age);
    return value;
  }).then(function(value){
    var result = confirm("Is the data correct?");
    return result;
  }).then(verify)
  .catch(function(reason){
    console.log(reason);
  });
```

Теперь код выглядит еще проще.

Метод Promise.resolve(value)

Метод `resolve()` объекта `Promise` принимает значение и преобразует

```
  console.log(value);
});
// переданный объект Promise
Promise.resolve(p1).then(function(value){
  console.log(value);
});
Promise.resolve({name: "Eden"}).then(function(value){
  console.log(value.name);
});
```

Результат выполнения:

```
4
4
Eden
```

Метод Promise.reject(value)

Метод `reject()` объекта `Promise` принимает значение и возвращает отклоненный объект `Promise` с указанным значением в качестве причины.

Метод `reject()`, в отличие от метода `Promise.resolve()`, используется для отладки, а не для преобразования значений в объекты `Promise`.

Следующий пример демонстрирует использование метода `reject()`:

```
var p1 = Promise.reject(4);
p1.then(null, function(value){
  console.log(value);
});
Promise.reject({name: "Eden"}).then(null, function(value){
  console.log(value.name);
});
```

Результат выполнения:

```
4
Eden
```

Метод Promise.all(iterable)

Метод `all()` объекта `Promise` принимает итерируемый объект в аргументе и возвращает объект `Promise`, когда все объекты `Promise` в итерируемом объекте будут выполнены.

Может пригодиться, когда необходимо выполнить какую-то задачу после завершения нескольких асинхронных операций.

Следующий пример демонстрирует использование метода `Promise.all()`:

```
var p1 = new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve();
  }, 1000);
});
var p2 = new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve();
  }, 2000);
});
var arr = [p1, p2];
Promise.all(arr).then(function(){
  console.log("Done"); // "Done" появится через 2 секунды
});
```

Если итерируемый объект содержит значение, не являющееся объектом `Promise`, оно преобразуется в объект `Promise` с помощью метода `Promise.resolve()`.

Если любой из переданных объектов `Promise` будет отклонен, метод `Promise.all()` немедленно вернет новый отклоненный объект `Promise` с той же причиной, что и у переданного отклоненного объекта `Promise`. Следующий пример демонстрирует это:

```
var p1 = new Promise(function(resolve, reject){
  setTimeout(function(){
    reject("Error");
  }, 1000);
});
var p2 = new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve();
  }, 2000);
});
var arr = [p1, p2];
Promise.all(arr).then(null, function(reason){
  console.log(reason); // "Error" появится через 1 секунду
});
```

Метод Promise.race(iterable)

Метод `race()` объекта `Promise` принимает итерируемый объект и возвращает объект `Promise` сразу, как только один объектов `Promise` в итерируемом объекте будет выполнен или отклонен с соответствующим результатом выполнения или причиной отклонения.

Название метода `race()` соответствует его действию, он устраивает гонки (race) между объектами `Promise` и отслеживает пришедшего первым.

Следующий пример демонстрирует использование метода `race()`:

```
var p1 = new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve("Fulfillment Value 1");
  }, 1000);
});
var p2 = new Promise(function(resolve, reject){
  setTimeout(function(){
    resolve("fulfillment Value 2");
  }, 2000);
});
var arr = [p1, p2];
Promise.race(arr).then(function(value){
  console.log(value); // Выведет "Fulfillment value 1"
}, function(reason){
  console.log(reason);
});
```

Программные интерфейсы JavaScript, основанные на объектах Promise

Новые асинхронные интерфейсы JavaScript основываются на объекте `Promise`, а не на событиях и обратных вызовах. Кроме того новые версии существующих программных интерфейсов JavaScript теперь так же основываются на объекте `Promise`.

Например, старые версии программных интерфейсов состояния батареи и веб-криптографии были основаны на событиях, а новые версии реализованы с помощью одних лишь объектов `Promise`. Давайте познакомимся с этими программными интерфейсами.

Программный интерфейс состояния батареи

Программный интерфейс состояния батареи позволяет контролировать текущий уровень заряда батареи и режим зарядки. Следующий пример демонстрирует новый программный интерфейс состояния заряда батареи:

```
navigator.getBattery().then(function(value){
  console.log("Batter Level: " + (value.level * 100));
    }, function(reason){
  console.log("Error: " + reason);
});
```

Метод `getBattery()` объекта `navigator` возвращает выполненный объект `Promise`, если информация о состоянии батареи успешно извлечена. В противном случае, возвращается отклоненный объект `Promise`.

Если объект `Promise` выполнен, результатом является объект с информацией о состоянии батареи. Свойство `level` этого объекта представляет уровень оставшегося заряда.

Программный интерфейс веб-криптографии

Программный интерфейс веб-криптографии позволяет выполнять хеширование, создание и проверку подписей, шифрование и дешифрование.

Следующий пример демонстрирует новый программный интерфейс веб-криптографии:

```
function convertStringToArrayBufferView(str)
{
  var bytes = new Uint8Array(str.length);
  for (var iii = 0; iii < str.length; iii++)
  {
    bytes[iii] = str.charCodeAt(iii);
  }
  return bytes;
}
function convertArrayBufferToHexaDecimal(buffer)
{
  var data_view = new DataView(buffer)
  var iii, len, hex = '', c;
  for(iii = 0, len = data_view.byteLength; iii < len; iii++)
  {
    c = data_view.getUint8(iii).toString(16);
    if(c.length < 2)
    {
      c = '0' + c;
    }
    hex += c;
  }
  return hex;
}
window.crypto.subtle.digest({name: "SHA-256"}, convertStringToArrayBuf
```

```
ferView("ECMAScript 6")).then(function(result){
  var hash_value = convertArrayBufferToHexaDecimal(result);
  console.log(hash_value);
});
```

В этом примере вычисляется хеш-код SHA-256 строки.

Метод `window.crypto.subtle.digest` принимает буферный массив со строкой и имя хеш-алгоритма, и возвращает объект `Promise`. Если хеширование увенчалось успехом, возвращается выполненный объект `Promise` с буферным массивом, содержащим хеш-код.

Итоги

В этой главе мы узнали, как JavaScript выполняет асинхронный код. Познакомились с различными моделями асинхронного кода. Рассмотрели, как объекты `Promise` облегчают чтение и разработку асинхронного кода, и как пользоваться программным интерфейсом Promise API спецификации ES6. Мы также узнали о некоторых прочих программных интерфейсах JavaScript, основанных на применении объекта `Promise`. В целом, глава была посвящена объектам `Promise`, их преимуществам и использованию программных интерфейсов, основанных на них.

В следующей главе мы познакомимся с программным интерфейсом Reflect API и особенностями его использования.

ГЛАВА 5. Реализация Reflect API

Спецификация ES6 вводит новый программный интерфейс Reflect API отражения объектов (позволяющий исследовать и обрабатывать свойства объектов). Хотя в спецификации ES5 уже присутствуют программные интерфейсы отражения объектов, они были организованы не лучшим образом и в случае ошибок возбуждали исключения. В спецификации ES6 программный интерфейс Reflect API организован намного лучше, облегчает чтение и разработку кода, а кроме того не возбуждает исключений в случае ошибок. Вместо этого он возвращает логическое значение, определяющее успех операции. Этот программный интерфейс широко используется разработчиками, поэтому важно знать его и уметь им пользоваться.

В этой главе мы рассмотрим:

- вызов функции с указанным значением ключевого слова `this`;
- вызов конструктора со свойством `prototype` другого конструктора;
- чтение и изменение атрибутов свойств объекта;
- перечисление свойств объекта с помощью итераторов;
- получение и установку внутреннего свойства `[[prototype]]` объекта;
- множество других операций исследования и манипулирования методами и свойствами объектов.

Объект Reflect

Глобальный объект `Reflect` в спецификации ES6 предоставляет новые методы для отражения объектов. Объект `Reflect` не является объектом-функцией, поэтому его нельзя вызвать. Кроме того, к нему нельзя применить оператор `new`.

Все методы программного интерфейса Reflect API в спецификации ES6 обернуты объектом `Reflect`, что делает его хорошо организованным.

Объект `Reflect` предоставляет множество методов, которые перекрывают методы глобального объекта `Object` с точки зрения функциональности.

Давайте рассмотрим методы, предоставляемые объектом `Reflect` для отражения объектов.

Метод Reflect.apply(function, this, args)

Метод `Reflect.apply()` используется для вызова функции с заданным значением ключевого слова `this`. Функция, вызываемая методом `Reflect.apply()`, называется целевой функцией. Этот метод действует подобно методу `apply()` объекта-функции.

Метод `Reflect.apply()` принимает три аргумента:

- *первый* аргумент представляет целевую функцию;
- *второй* необязательный аргумент представляет значение ключевого слова `this` внутри целевой функции;
- *третий* необязательный аргумент является объектом массива с аргументами целевой функции.

Метод `Reflect.apply()` возвращает все, что вернет целевая функция.

Следующий пример демонстрирует использование метода `Reflect.apply()`:

```
function function_name(a, b, c)
{
  return this.value + a + b + c;
}
var returned_value = Reflect.apply(function_name, {value: 100},
[10, 20, 30]);
console.log(returned_value); // Выведет "160"
```

Метод Reflect.construct(constructor, args, prototype)

Метод `Reflect.construct()` используется для вызова функции в качестве конструктора. Он действует подобно оператору `new`. Функция, которая будет использоваться в роли конструктора, называется **целевым конструктором** (target constructor).

Единственной веской причиной использования метода `Reflect`

лении свойства объекта возникнет ошибка, он возбудит исключение, тогда как `Reflect.defineProperty()` вернет результат `false`.

Метод `Reflect.defineProperty()` принимает три аргумента:

- *первый* аргумент представляет объект, чье свойство должно быть определено или изменено;
- *второй* аргумент представляет символ или имя свойства, которое должно быть определено или изменено;
- *третий* аргумент представляет дескриптор свойства, которое определяется или изменяется.

Понятия свойств данных и свойств со средствами доступа

Согласно спецификации ES5 каждое свойство объекта может быть либо свойством данных, либо свойством со средствами доступа. Свойство данных имеет значение, которое может быть, а может и не быть, доступным для записи, тогда как свойство со средствами доступа снабжено парой функций для установки и извлечения значений свойства.

Атрибутами свойства данных являются значение, признак доступности для записи, признак доступности для перечисления и признак доступности для настройки. С другой стороны, атрибутами свойства со средствами доступа являются `set`, `get`, `enumerable` и `configurable`.

Дескриптор — это объект, описывающий атрибуты свойства. При

- `enumerable`: если равно `true`, свойство может быть использовано в цикле `for...in` и в методе `Object.keys()`. Значение по умолчанию `false`.

Дескриптор свойства со средствами доступа имеет следующие свойства:

- `get`: функция, возвращающая значение свойства. Эта функция не имеет параметров, значение по умолчанию не определено.
- `set`: функция, устанавливающая значение свойства. Этой функции будет передан аргумент с новым значением для присваивания свойству.
- `configurable`: если равно `true`, свойства дескриптора могут быть изменены и свойство может быть удалено. Значение по умолчанию `false`.
- `enumerable`: если равно `true`, свойство может быть использовано в цикле `for...in` и в методе `Object.keys()`. Значение по умолчанию `false`.

В зависимости от свойств объекта дескриптора, JavaScript решает, является ли свойство свойством данных или свойством со средствами доступа.

Если свойство добавляется без помощи методов `Reflect.defineProperty()`, `Object.defineProperty()`, `Object.defineProperties()` или `Object.create()`, создается свойство данных, а его атрибуты `writable`, `enumerable` и `configurable` получают значение `true`. После добавления свойства, вы можете изменить его атрибуты.

Если к моменту вызова `Reflect.defineProperty()`, `Object.defineProperty()` или `Object.defineProperties()` объект уже имеет свойство с указанным именем, существующее свойство будет изменено. Атрибуты, не указанные в дескрипторе, сохранят прежние значения.

Свойство данных можно превратить в свойство со средствами доступа, и наоборот. В этом случае атрибуты `configurable` и `enumerable`, не указанные в дескрипторе, сохранят прежние значения. Другие атрибуты, не указанные в дескрипторе, получат значения по умолчанию.

Следующий пример демонстрирует создание свойства данных с помощью метода `Reflect.defineProperty()`:

```
var obj = {}
Reflect.defineProperty(obj, "name", {
  value: "Eden",
  writable: true,
  configurable: true,
  enumerable: true
});
console.log(obj.name); // Выведет "Eden"
```

А следующий пример демонстрирует создание свойства со средствами доступа с помощью метода `Reflect.defineProperty()`:

```
var obj = {
  __name__: "Eden"
}
Reflect.defineProperty(obj, "name", {
  get: function(){
    return this.__name__;
  },
  set: function(newName){
    this.__name__ = newName;
  },
  configurable: true,
  enumerable: true
});
obj.name = "John";
console.log(obj.name); // Выведет "John"
```

Метод Reflect.deleteProperty(object, property)

Метод `Reflect.deleteProperty()` используется для удаления свойства объекта. Он действует подобно оператору `delete`.

Этот метод принимает два аргумента: первый аргумент представляет ссылку на объект, а второй – имя свойства, подлежащего удалению. Метод `Reflect.deleteProperty()` возвращает `true`, если свойство успешно удалено. В противном случае возвращается `false`.

Следующий пример демонстрирует удаление свойства с помощью метода `Reflect.deleteProperty()`:

```
var obj = {
  name: "Eden"
}
console.log(obj.name); // Выведет "Eden"
Reflect.deleteProperty(obj, "name");
console.log(obj.name); // Выведет "undefined"
```

Метод Reflect.enumerate(object)

Метод `Reflect.enumerate()` принимает объект и возвращает объект итератора, представляющий перечисляемые свойства объекта. Он также возвращает унаследованные перечисляемые свойства объекта.

Этот метод действует подобно циклу `for...in`, с той лишь разницей, что `Reflect.enumerate()` возвращает итератор, а цикл `for...in` выполняет обход все перечислимых свойств.

Следующий пример демонстрирует использование метода `Reflect.enumerate()`:

```
var obj = {
  a: 1,
  b: 2,
  c: 3
};
var iterator = Reflect.enumerate(obj);
console.log(iterator.next().value);
console.log(iterator.next().value);
```

```
});
console.log(obj.name); // Выведет "Eden"
var name = Reflect.get(obj, "name", {__name__: "John"});
console.log(name); // Выведет "John"
```

Метод Reflect.set(object, property, value, this)

Метод `Reflect.set()` используется для установки значения свойства объекта. Первый аргумент представляет объект, второй – имя свойства, а третий – новое значение свойства. Если свойство обладает средствами доступа, можно передать четвертый аргумент, определяющий значение ключевого слова `this` внутри функции `set`.

Метод `Reflect.set()` возвращает `true`, если свойство успешно установлено. В противном случае возвращается `false`.

Следующий пример демонстрирует использование метода `Reflect.set()`:

```
var obj1 = {
  __name__: "Eden"
};
Reflect.defineProperty(obj1, "name", {
  set: function(newName){
    this.__name__ = newName;
  },
  get: function(){
    return this.__name__;
  }
});
var obj2 = {
  __name__: "John"
};
Reflect.set(obj1, "name", "Eden", obj2);
console.log(obj1.name); // Выведет "Eden"
console.log(obj2.__name__); // Выведет "Eden"
```

Метод Reflect.getOwnPropertyDescriptor (object, property)

Метод `Reflect.getOwnPropertyDescriptor()` используется для получения дескриптора свойства объекта.

Он действует подобно методу `Object.getOwnPropertyDescriptor()`.

Метод `Reflect.getOwnPropertyDescriptor()` принимает два аргумента. В первом передается объект, а во втором – имя свойства.

Следующий пример демонстрирует использование метода `Reflect.getOwnPropertyDescriptor()`:

```
var obj = {
  name: "Eden"
};
var descriptor = Reflect.getOwnPropertyDescriptor(obj, "name");
console.log(descriptor.value);
console.log(descriptor.writable);
console.log(descriptor.enumerable);
console.log(descriptor.configurable);
```

Результат выполнения:

```
Eden
true
true
true
```

Метод Reflect.getPrototypeOf(object)

Метод `Reflect.getPrototypeOf()` используется для извлечения прототипа объекта, то есть, значения внутреннего свойства `[[prototype]]` объекта.

Метод `Reflect.getPrototypeOf()` полностью аналогичен методу `Object.getPrototypeOf()`.

Следующий пример демонстрирует использование метода `Reflect.getPrototypeOf()`:

```
var obj1 = {
  __proto__: {
    name: "Eden"
  }
};
var obj2 = Reflect.getPrototypeOf(obj1);
console.log(obj2.name); // Выведет "Eden"
```

Метод Reflect.setPrototypeOf(object, prototype)

Метод `Reflect.setPrototypeOf()` используется для установки значения внутреннего свойства объекта `[[prototype]]`. Возвращает `true`, если внутреннее свойство `[[prototype]]` было успешно установлено. В противном случае возвращается `false`.

Следующий пример демонстрирует использование метода:

```
var obj = {};
Reflect.setPrototypeOf(obj, {
  name: "Eden"
```

```
});
console.log(obj.name); // Выведет "Eden"
```

Метод Reflect.has(object, property)

Метод `Reflect.has()` используется для проверки существования свойства в объекте. Он также проверяет существование унаследованных свойств. Возвращает `true`, если свойство существует. В противном случае возвращает `false`.

Метод полностью аналогичен оператору `in`.

Следующий пример демонстрирует использование метода `Reflect.has()`:

```
var obj = {
  __proto__: {
    name: "Eden"
  },
  age: 12
};
console.log(Reflect.has(obj, "name")); // Выведет "true"
console.log(Reflect.has(obj, "age")); // Выведет "true"
```

Метод Reflect.isExtensible(object)

Метод `Reflect.isExtensible()` проверяет возможность расширения объекта, то есть, возможность добавления новых свойств в объект.

Объект может быть отмечен как нерасширяемый с помощью методов `Object.preventExtensions()`, `Object.freeze()` и `Object.seal()`.

Метод `Reflect.isExtensible()` полностью аналогичен методу `Object.isExtensible()`.

Следующий пример демонстрирует использование метода `Reflect.isExtensible()`:

```
var obj = {
  name: "Eden"
};
console.log(Reflect.isExtensible(obj)); // Выведет "true"
Object.preventExtensions(obj);
console.log(Reflect.isExtensible(obj)); // Выведет "false"
```

Метод Reflect.preventExtensions(object)

Метод `Reflect.preventExtensions()` позволяет отметить объект как нерасширяемый. Возвращает логическое значение, указывающее, была ли операция выполнена успешно или нет.

Он полностью аналогичен методу `Object.preventExtensions()`:

```
var obj = {
  name: "Eden"
};
console.log(Reflect.isExtensible(obj)); // Выведет "true"
console.log(Reflect.preventExtensions(obj)); // Выведет "true"
console.log(Reflect.isExtensible(obj)); //Output "false"
```

Метод Reflect.ownKeys(object)

Метод `Reflect.ownKeys()` возвращает массив, элементы которого представляют ключи свойств переданного объекта. Метод игнорирует наследуемые свойства.

Следующий пример демонстрирует использование метода:

```
var obj = {
  a: 1,
  b: 2,
  __proto__: {
    c: 3
  }
};
var keys = Reflect.ownKeys(obj);
console.log(keys.length); // Выведет "2"
console.log(keys[0]); // Выведет "a"
console.log(keys[1]); //Output "b"
```

Итоги

В этой главе мы познакомились со средствами отражения объектов и с особенностями использования программного интерфейса Reflect API из спецификации ES6. Мы рассмотрели методы объекта `Reflect` с примерами их использования. В целом эта глава была посвящена применению программного интерфейса Reflect API из спецификации ES6 для чтения и изменения свойств объектов.

В следующей главе мы познакомимся с прокси-объектами из спецификации ES6 и их использованием.

ГЛАВА 6. Использование прокси-объектов

Прокси-объекты используется для определения нестандартного поведения основных операций над объектами. Подобные объекты существуют в таких языках программирования, как C#, C++ и Java, но в JavaScript никогда ранее не было прокси-объектов. Спецификация ES6 предлагает программный интерфейс Proxy API для создания прокси-объектов. В этой главе мы рассмотрим прокси-объекты, их использование и ловушки. Все больше разработчиков признают преимущества прокси-объектов и активно их используют, поэтому важно узнать о прокси-объектах и примерах его использования как можно больше, что мы и сделаем в этой главе.

В этой главе мы рассмотрим:

- создание прокси-объектов с помощью программного интерфейса Proxy API;
- понимание прокси-объектов и их использования;
- перехват различных операций с помощью ловушек;
- различные виды доступных ловушек;
- некоторые частные случаи использование прокси-объектов.

Основы прокси-объектов

Прокси-объект действует подобно обертке и определяет нестандартное поведение при выполнении основных операций над объектом. К таким основным операциям относятся: поиск свойств, присваивание значений свойствам, вызов конструктора, перечисление и так далее.

После того, как объект будет завернут в прокси-объект, все операции над объектом фактически будут производиться над прокси-объектом, что позволяет реализовать нестандартное поведение операций.

Терминология

Вот некоторые важные термины, которые будут использованы в процессе знакомства с прокси-объектами:

- **целевой объект**: объект, завернутый в прокси-объект;
- **ловушки**: функции, перехватывающие различные операции над целевым объектом, и определяющие нестандартное поведение этих операций;
- **обработчик**: объект, содержащий ловушки и прикрепленный к прокси-объекту.

Программный интерфейс Proxy API

Программный интерфейс Proxy API в спецификации ES6 включает конструктор `Proxy` для создания прокси-объектов. Он принимает два аргумента:

- **целевой объект**: объект, который будет завернут в прокси-объект;
- **обработчик**: объект с ловушками для целевого объекта.

Ловушки можно определять для любых возможных операций над целевым объектом. Если для какой-то операции ловушка не определена, выполняется действие по умолчанию.

Следующий пример демонстрирует создание прокси-объекта и выполнение операций над целевым объектом. В этом примере не определяется ни одной ловушки:

```
var target = {
  age: 12
};
var handler = {};
var proxy = new Proxy(target, handler);
proxy.name = "Eden";
console.log(target.name);
console.log(proxy.name);
console.log(target.age);
console.log(proxy.age);
```

Результат выполнения:

```
Eden
Eden
12
12
```

Как видите, свойство `age` целевого объекта `target` можно получить посредством объекта `proxy`. При добавлении свойства `name` в объект `proxy`, оно фактически добавляется в объект `target`.

Так как для операции присваивания не была определена ловушка, при попытке присвоить свойству `proxy.name` новое значение, была выполнена операция по умолчанию – простое присваивание.

Итак, можно сказать, что `proxy` является оберткой вокруг объекта `target`, а ловушки помогают изменить поведение операций.

Многие разработчики не сохраняют ссылку на целевой объект, используя доступа к нему прокси-объект. Ссылка необходима, только когда нужно использовать ее для нескольких прокси-объектов. Предыдущий код можно переписать иначе:

```
var proxy = new Proxy({
  age: 12
}, {});
proxy.name = "Eden";
```

Ловушки

Существует несколько видов ловушек для различных операций, выполняемых над объектом. Некоторые виды ловушек должны возвращать значения. Имеются некоторые правила, которым они должны следовать, возвращая значения. Возвращаемые значения перехватываются прокси-объектом для фильтрации и, возможно, для проверки соответствия возвращаемых значений заданным правилам. Если возвращаемое ловушкой значение противоречит правилам, прокси-объект возбудит исключение `TypeError`. Ключевое слово `this` внутри ловушки всегда ссылается на обработчик. Давайте рассмотрим различные виды ловушек.

Метод get(target, property, receiver)

Ловушка `get` вызывается, когда выполняется попытка получить значение свойства с помощью операторов точки и квадратных скобок, или метода `Reflect.get()`. Она принимает три параметра: целевой объект, имя свойства и прокси-объект, и должна возвращать значение свойства.

Следующий пример демонстрирует использование ловушки `get`:

```
var proxy = new Proxy({
  age: 12
}, {
  get: function(target, property, receiver){
    if(property in target)
    {
      return target[property];
    }
    else
    {
      return "Not Found";
    }
  }
}
);
console.log(Reflect.get(proxy, "age"));
console.log(Reflect.get(proxy, "name"));
```

Результат выполнения кода:

```
12
Not found
```

Здесь ловушка `get` пытается найти свойство объекта `target` и, если находит, возвращает его значение. В противном случае возвращает строку, сообщающую, что свойство не найдено.

Параметр `receiver` представляет ссылку на объект, свойство которого мы намереваемся получить. Рассмотрим следующий пример, чтобы лучше понять сущность параметра `receiver`:

```
var proxy = new Proxy({age: 13}, {
  get: function(target, property, receiver){
    console.log(receiver);
    if(property in target)
    {
      console.log(receiver);
      return target[property];
    }
    else
    {
      return "Not Found";
    }
  }
}
);
var temp = proxy.name;
var obj = {
  age: 12,
  __proto__: proxy
```

```
}
temp = obj.name;
```

Результат выполнения:

```
{age: 13}
{age: 12}
```

Здесь объект `obj` наследует свойства объекта `proxy`. Поэтому, когда свойство `name` не обнаружилось в объекте `obj`, поиск был продолжен в объекте `proxy`. Так как объект `proxy` обладает ловушкой `get`, именно она и вернула значение.

То есть, значением параметра `receiver` при обращении к свойству `obj.name` была ссылка на `obj`, а при обращении к свойству `proxy.name` — ссылка на `proxy`.

Значение параметра `receiver` одинаково определяется для всех других видов ловушек.

Правила

При использовании ловушки `get` должны соблюдаться следующие правила:

- возвращаемое значение должно совпадать со значением свойства целевого объекта, если свойство целевого объекта является свойством данных и недоступно для записи или настройки;
- возвращаемое значение должно быть `undefined`, если свойство целевого объекта не является настраиваемым свойством со средствами доступа и имеет атрибут `[[Get]]` со значением `undefined`.

Метод set(target, property, value, receiver)

Ловушка `set` вызывается при попытке изменить значение свойства с помощью оператора присваивания или метода `Reflect.set()`. Ей передаются четыре параметра: целевой объект, имя свойства, новое значение и приемник.

Ловушка `set` должна вернуть `true`, если присваивание выполнено успешно, или `false`, в противном случае.

Следующий пример демонстрирует использование ловушки `set`:

```
var proxy = new Proxy({}, {
  set: function(target, property, value, receiver){
    target[property] = value;
    return true;
```

```
  }
});
Reflect.set(proxy, "name", "Eden");
console.log(proxy.name); // Выведет "Eden"
```

Правила

При использовании ловушки `set` должны соблюдаться следующие правила:

- если свойство целевого объекта является свойством данных и недоступно для записи или настройки, возвращается значение `false`, так как нельзя изменить значение этого свойства;
- если свойство целевого объекта не является настраиваемым свойством со средствами доступа и имеет атрибут `[[Set]]` со значением `undefined`, возвращается значение `false`, так как нельзя изменить значение свойства.

Метод has(target, property)

Ловушка `has` вызывается при проверке существования свойства с помощью оператора `in`. Она принимает два параметра: целевой объект и имя свойства, и должна возвращать логическое значение, определяющее существование свойства.

Следующий пример демонстрирует использование ловушки `has`:

```
var proxy = new Proxy({age: 12}, {
  has: function(target, property){
    if(property in target)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
});
console.log(Reflect.has(proxy, "name"));
console.log(Reflect.has(proxy, "age"));
```

Результат выполнения:

```
false
true
```

Правила

При использовании ловушки `has` должны соблюдаться следующие правила:

- нельзя вернуть `false`, если свойство существует как недоступное для настройки собственное свойство объекта `target`;
- нельзя вернуть `false`, если свойство существует как собственное свойство целевого объекта и целевой объект не является расширяемым.

Метод isExtensible(target)

Ловушка `isExtensible` выполняется, когда проверяется возможность расширения объекта с помощью метода `Object.isExtensible()`. Ей передается всего один параметр – целевой объект. Она должна возвращать логическое значение, определяющее возможность расширения объекта.

Следующий пример демонстрирует использование ловушки `IsExtensible`:

```
var proxy = new Proxy({age: 12}, {
  isExtensible: function(target){
    return Object.isExtensible(target);
  }
});
console.log(Reflect.isExtensible(proxy)); // Выведет "true"
```

Правила

При использовании ловушки `isExtensible` должно соблюдаться следующее правило:

- нельзя вернуть `false`, если целевой объект является расширяемым. Аналогично, нельзя вернуть `true`, если целевой объект не является расширяемым;

Метод getPrototypeOf(target)

Ловушка `getPrototypeOf` выполняется при попытке получить значение внутреннего свойства `[[prototype]]` с помощью метода `Object.getPrototypeOf()` или свойства `__proto__`. Ей передается всего один параметр – целевой объект.

Она должна вернуть объект или значение `null`. Значение `null` указывает, что объект ничего не наследует и находится в начале иерархии наследования.

Следующий пример демонстрирует использование ловушки `getPrototypeOf`:

```
var proxy = new Proxy({age: 12, __proto__: {name: "Eden"}}, {
  getPrototypeOf: function(target){
    return Object.getPrototypeOf(target);
  }
});
console.log(Reflect.getPrototypeOf(proxy).name); // Выведет "Eden"
```

Правила

При использовании ловушки `getPrototypeOf` должны соблюдаться следующие правила:

- ловушка должна вернуть объект или значение `null`;
- если целевой объект не является расширяемым, ловушка должна вернуть фактический прототип.

Метод setPrototypeOf(target, prototype)

Ловушка `setPrototypeOf` выполняется при попытке изменить значение внутреннего свойства `[[prototype]]` с помощью метода `Object.setPrototypeOf()` или свойства `__proto__`. Она принимает два параметра: целевой объект и новое значение свойства.

Эта ловушка возвращает логическое значение, сообщающее об успехе изменения прототипа.

Следующий пример демонстрирует использование ловушки `setPrototypeOf`:

```
var proxy = new Proxy({}, {
  setPrototypeOf: function(target, value){
    Reflect.setPrototypeOf(target, value);
    return true;
  }
});
Reflect.setPrototypeOf(proxy, {name: "Eden"});
console.log(Reflect.getPrototypeOf(proxy).name); // Выведет "Eden"
```

Правила

При использовании ловушки `setPrototypeOf` должно соблюдаться

```
console.log("Configurable: " + descriptor.configurable);
console.log("Value: " + descriptor.value);
```

Результат выполнения:

```
Enumerable: true
Writable: true
Configurable: true
Value: 12
```

Правила

При использовании ловушки `getOwnPropertyDescriptor` должны соблюдаться следующие правила:

- ловушка должна возвращать объект или `undefined`;
- нельзя вернуть значение `undefined`, если свойство существует и является недоступным для настройки собственным свойством объекта `target`;
- нельзя вернуть значение `undefined`, если свойство существует и является собственным свойством объекта `target`, а объект `target` является нерасширяемым;
- должно возвращаться значение `undefined`, если не существует собственного свойства объекта `target` и объект `target` является нерасширяемым;
- нельзя присвоить свойству `configurable` возвращаемого объекта дескриптора значение `false`, если указанное свойство является собственным свойством объекта `target` или настраиваемым собственным свойством объекта `target`.

Метод defineProperty(target, property, descriptor)

Ловушка `defineProperty` выполняется при попытке определить свойство с помощью метода `Object.defineProperty()`. Она принимает три параметра: целевой объект, имя свойства и объект дескриптора.

Ловушка должна возвращать логическое значение, указывающее на успех операции определения свойства.

Следующий пример демонстрирует использование ловушки `defineProperty`:

```
var proxy = new Proxy({}, {
  defineProperty: function(target, property, descriptor){
    Object.defineProperty(target, property, descriptor);
    return true;
```

```
  }
});
Reflect.defineProperty(proxy, "name", {value: "Eden"});
console.log(proxy.name); // Выведет "Eden"
```

Правила

При использовании ловушки `defineProperty` должно соблюдаться следующее правило:

- ловушка должна возвращать `false`, если объект `target` не является расширяемым или свойство еще не существует.

Метод deleteProperty(target, property)

Ловушка `deleteProperty` выполняется при попытке удалить свойство с помощью оператора `delete` или метода `Reflect.deleteProperty()`. Она принимает два параметра: целевой объект и имя свойства.

Ловушка должна возвращать логическое значение, указывающее на успех операции удаления.

Следующий пример демонстрирует использование ловушки `deleteProperty`:

```
var proxy = new Proxy({age: 12}, {
  deleteProperty: function(target, property){
    return delete target[property];
  }
});
Reflect.deleteProperty(proxy, "age");
console.log(proxy.age); // Выведет "undefined"
```

Правила

При использовании ловушки `deleteProperty` должно соблюдаться следующее правило:

- ловушка должна вернуть `false`, если свойство является недоступным для настройки собственным свойством объекта `target`.

Метод enumerate(target)

Ловушка `enumerate` выполняется, когда производится обход ключей свойств помощью цикла `for...in` или метода `Reflect.enumerate()`. Она принимает один параметр, объект `target`.

Ловушка должна возвращать итератор, содержащий ключи перечисляемых свойств объекта.

Следующий пример демонстрирует использование ловушки `enumerate`:

```
var proxy = new Proxy({age: 12, name: "Eden"}, {
  enumerate: function(target){
    var arr = [];
    for(var p in target)
    {
      arr[arr.length] = p;
    }
    return arr[Symbol.iterator]();
  }
});
var iterator = Reflect.enumerate(proxy);
console.log(iterator.next().value);
console.log(iterator.next().value);
console.log(iterator.next().done);
```

Результат выполнения:

```
age
name
true
```

Правила

При использовании ловушки `enumerate` должно соблюдаться следующее правило:

- ловушка должна возвращать объект.

Метод ownKeys(target)

Ловушка `ownKeys` выполняется, когда при извлечении ключей собственных свойств с помощью метода `Reflect.ownKeys()`, `Object.getOwnPropertyNames()`, `Object.getOwnPropertySymbols()` или `Object.keys()`. Она принимает всего один параметр – объект `target`.

Метод `Reflect.ownKeys()` подобен методу `Object.getOwnPropertyNames()`, то есть, возвращает ключи перечисляемых и не перечисляемых свойств объекта. Методы игнорируют наследуемые свойства. Разница состоит в том, что метод `Reflect.ownKeys()` возвращает как символьные, так и строковые ключи, в то время как метод `Object.getOwnPropertyNames()` возвращает только строковые ключи.

Метод `Object.getOwnPropertySymbols()` возвращает символьные ключи перечисляемых и не перечисляемых свойств, и игнорирует унаследованные свойства.

Метод `Object.keys()` действует подобно методу `Object.getOwnPropertyNames()`, с той лишь разницей, что `Object.keys()` возвращает только перечисляемые свойства.

Ловушка `ownKeys` должна вернуть массив с множеством ключей собственных свойств.

Следующий пример демонстрирует использование ловушки `ownKeys`:

```
var s = Symbol();
var object = {age: 12, __proto__: {name: "Eden"}, [s]: "Symbol"};
Object.defineProperty(object, "profession", {
  enumerable: false,
  configurable: false,
  writable: false,
  value: "Developer"
})
var proxy = new Proxy(object, {
  ownKeys: function(target){
    return Object.getOwnPropertyNames(target).concat(
    Object.getOwnPropertySymbols(target));
  }
});
console.log(Reflect.ownKeys(proxy));
console.log(Object.getOwnPropertyNames(proxy));
console.log(Object.keys(proxy));
console.log(Object.getOwnPropertySymbols(proxy));
```

Результат выполнения:

```
["age", "profession", Symbol()]
["age", "profession"]
["age"]
[Symbol()]
```

Здесь элементы массива, возвращаемого ловушкой `ownKeys`, фильтруются прокси-объектом для получения нужного результата. Например, при вызове метода `Object.getOwnPropertySymbols()` возвращается массив только символов. Следовательно, прокси-объект удалил строчные ключи из возвращаемого массива.

Правила

При использовании ловушки `ownKeys` должны соблюдаться следующие правила:

- элементы возвращаемого массива должны быть либо строками, либо символами;

- возвращаемый массив должен содержать ключи всех недоступных для настройки собственных свойств объекта `target`;
- если объект `target` является нерасширяемым, возвращаемый массив должен содержать все ключи собственных свойств объекта `target` и никаких других значений.

Метод apply(target, thisValue, arguments)

Если целевой объект является функцией, прокси-объект будет вызывать ловушку `apply`. Ловушка `apply` также выполняется для методов функций `apply()` и `call()`, и метода `Reflect.apply()`.

Ловушка `apply` принимает три параметра. Первый параметр представляет объект `target`, третий – массив с аргументами вызываемой функции. Во втором параметре передается значение `this` целевой функции, то есть, то же самое значение, которое было бы передано целевой функции при вызове без участия прокси.

Следующий пример демонстрирует использование ловушки `apply`:

```
var proxy = new Proxy(function(){}, {
  apply: function(target, thisValue, arguments){
    console.log(thisValue.name);
    return arguments[0] + arguments[1] + arguments[2];
  }
});
var obj = {
  name: "Eden",
  f: proxy
}
var sum = obj.f(1,2,3);
console.log(sum);
```

Результат выполнения:

```
Eden
6
```

Метод construct(target, arguments)

Если целевой объект является функцией, ловушка `construct` будет вызвана при выполнении конструктора объекта с помощью оператора `new` или метода `Reflect.construct()`.

Ловушка `construct` принимает два параметра. Первый параметр представляет объект `target`, а второй – массив аргументов конструктора.

Ловушка `construct` должна возвращать вновь созданный экземпляр.

Следующий пример демонстрирует использование ловушки `construct`:

```
var proxy = new Proxy(function(){}, {
  construct: function(target, arguments){
    return {name: arguments[0]};
  }
});
var obj = new proxy("Eden");
console.log(obj.name); // Выведет "Eden"
```

Метод Proxy.revocable(target, handler)

Отзываемыми называются прокси-объекты, которые могут быть отозваны (то есть отключены).

Для создания отзываемых прокси-объектов требуется использовать метод `Proxy.revocable()`. Метод `Proxy.revocable()` не является конструктором. Он принимает те же аргументы, что и конструктор `Proxy`, но вместо отзываемого экземпляра прокси-объекта возвращает объект с двумя следующими свойствами:

- `proxy`: отзываемый объект прокси;
- `revoke`: функция, отключающая `proxy`.

После отзыва прокси-объекта любые попытки его использования будут возбуждать исключение `TypeError`.

Следующий пример демонстрирует создание отзываемого прокси-объект и его отзыва:

```
var revocableProxy = Proxy.revocable({
  age: 12
}, {
  get: function(target, property, receiver){
    if(property in target)
    {
      return target[property];
    }
    else
    {
      return "Not Found";
    }
  }
}
);
```

```
console.log(revocableProxy.proxy.age);
revocableProxy.revoke();
console.log(revocableProxy.proxy.name);
```

Результат выполнения кода:

```
12
TypeError: proxy is revoked
```

Возможный сценарий использования

Отзываемый прокси-объект можно использовать вместо обычного прокси-объекта, когда предполагается передать прокси-объект в функцию, которая выполняется асинхронно или параллельно, отозвав его, если в функции этот прокси-объект не должен использоваться.

Использование прокси

Ниже перечислены некоторые основные случаи использования прокси-объектов:

- создание виртуальных объектов: удаленных объектов, хранимых объектов и так далее;
- отложенное создание объектов;
- прозрачное журналирование, трассировка, профилирование и многое другое;
- специализированные языки для встраиваемых окружений;
- встраиваемые абстракции для контроля доступа.

Итоги

В этой главе мы познакомились с прокси-объектами и их использованием. Рассмотрели различные ловушки вместе с примерами их использования. Мы также узнали о правилах, которые должны соблюдать ловушки. Эта глава была посвящена глубокому изучению программного интерфейса Proxy API в спецификации ES6. Наконец, мы узнали об областях применения прокси-объектов.

В следующей главе, мы рассмотрим объектно-ориентированное программирование и классы в спецификации ES6.

ГЛАВА 7. Прогулка по классам

Спецификация ES6 вводит классы, которые обеспечивают простой и понятный синтаксис наследования и создания конструкторов. В языке JavaScript никогда не существовало понятия классов, хотя это и объектно-ориентированный язык программирования. Программистам, имеющим опыт работы с другими языками программирования, трудно разобраться в объектно-ориентированной модели и наследовании JavaScript из-за отсутствия классов. В этой главе, мы изучим объектно-ориентированную модель JavaScript, построенную на кассах из спецификации ES6:

- типы данных JavaScript;
- создание объектов классическим способом;
- конструкторы примитивных типов;
- какие классы существуют в ES6;
- создание объектов с помощью классов;
- наследование в классах;
- особенности классов.

Понимание объектно-ориентированной модели JavaScript

Прежде чем перейти к классам в ES6, освежим наши знания о типах данных, конструкторах и наследовании в JavaScript. Изучая классы, мы сравним синтаксис конструкторов и наследования на базе прототипов с синтаксисом классов. Для этого понадобится хорошее знание этой темы.

Типы данных JavaScript

Переменные JavaScript содержат (или хранят) данные (или значения). Тип хранимых данных называется **типом данных**. В JavaScript существует семь типов данных: **number**, **string**, **Boolean**, **null**, **undefined**, **symbol** и **object**.

Что касается объектов – переменные хранят ссылки на объекты (то есть, адреса в памяти), а не сами объекты.

Все типы данных, кроме объектов, называются **элементарными типами данных**.

Массивы и функции фактически являются объектами JavaScript.

Создание объектов

В JavaScript существует два способа создания объектов: с помощью литерала объекта или конструктора. Литерал объекта используется для создания фиксированных объектов, а конструктор – для создания объектов во время выполнения.

Рассмотрим случай, когда необходимо использовать конструктор вместо литерала объекта:

```
var student = {
  name: "Eden",
  printName: function(){
    console.log(this.name);
  }
}
student.printName(); // Выведет "Eden"
```

Здесь объект `student` создается с помощью литерала объекта, то есть с использование формы записи в фигурных скобках `{}`. Этот способ прекрасно подходит для создания одного объекта `student`.

Но, если потребуется создать множество объектов `student`, такой подход может превратиться в проблему. Мало кому понравится повторять предыдущий код несколько раз, чтобы создать несколько объектов `student`. Это тот случай, когда требуется применить конструктор.

Когда функция вызывается с ключевым словом `new`, она действует как конструктор. Конструктор создает и возвращает объект. Ключевое слово `this` внутри функции, при вызове в качестве конструктора, указывает на новый экземпляр объекта, и как только конструктор за-

вершит выполнение, он автоматически вернет новый объект. Рассмотрим пример:

```
function Student(name)
{
  this.name = name;
}
Student.prototype.printName = function(){
  console.log(this.name);
}
var student1 = new Student("Eden");
var student2 = new Student("John");
student1.printName(); // Выведет "Eden"
student2.printName(); // Выведет "John"
```

Чтобы создать несколько объектов `student`, мы несколько раз вызвали конструктор, вместо использования нескольких литералов объекта.

Чтобы добавить методы, мы, вместо ключевого слово `this`, воспользовались свойством `prototype` конструктора. Почему мы поступили именно так, и о свойстве `prototype` рассказывается в следующем разделе.

В действительности каждый объект должен иметь конструктор. Каждый объект имеет унаследованное свойство `constructor`, содержащее ссылку на конструктор. Когда объект создается с помощью литерала, в его свойство `constructor` записывается ссылка на конструктор глобального объекта `Object`. Рассмотрим следующий пример, иллюстрирующий это:

```
var student = {}
console.log(student.constructor == Object); // Выведет "true"
```

Понятие наследования

Каждый объект JavaScript имеет внутреннее свойство `[[prototype]]` со ссылкой на другой объект, называющийся прототипом. Прототип, как и любой объект, имеет свой прототип, и так далее, пока не будет получен объект с прототипом `null`. Величина `null` не имеет прототипа, она является конечным звеном в цепи прототипов.

При попытке получить свойство, отсутствующее в объекте, его поиск будет продолжен в прототипе объекта. Если свойство не будет найдено в прототипе, будет выполнен поиск в прототипе прототипа, и так до тех пор, пока по цепи прототипов не будет достигнуто значение `null`. Именно так работает наследование в JavaScript.

Так как объект JavaScript может иметь только один прототип, JavaScript поддерживает только одиночное наследование.

При создании объектов с помощью литералов объектов можно использовать специальное свойство `__proto__` или метод `Object.setPrototypeOf()` для назначения объекту прототипа. JavaScript также предоставляет метод `Object.create()` для создания нового объекта с заданным прототипом в свойстве `__proto__`, который, впрочем, поддерживается не всеми браузерами. Можно также использовать метод `Object.setPrototypeOf()`, выглядящий менее странно. Следующий пример демонстрирует различные способы задания прототипа при создании объекта с помощью литерала объекта:

```
var object1 = {
name: "Eden",
  __proto__: {age: 24}
}
var object2 = {name: "Eden"}
Object.setPrototypeOf(object2, {age: 24});
var object3 = Object.create({age: 24}, {name: {value: "Eden"}});
console.log(object1.name + " " + object1.age);
console.log(object2.name + " " + object2.age);
console.log(object3.name + " " + object3.age);
```

Результат выполнения:

```
Eden 24
Eden 24
Eden 24
```

Объект `{age:24}` в примере выше называется **базовым объектом**, **суперобъектом** или **родительским объектом**, так как является родоначальником. Объект `{name:"Eden"}` называется **производным объектом**, **подобъектом** или **дочерним объектом**, так как он наследует другой объект.

Если не назначить прототип объекта при создании с помощью литерала, его свойство `prototype` будет ссылаться на свойство `Object.prototype`. Прототип `Object.prototype` равен `null`, так как является концом в цепи прототипов. Следующий пример иллюстрирует это:

```
var obj = {
  name: "Eden"
}
console.log(obj.__proto__ == Object.prototype); // Выведет "true"
```

При создании объектов с помощью конструктора, прототип новых объектов всегда указывает на свойство `prototype` объекта функции.

По умолчанию, свойство `prototype` хранит ссылку на объект с единственным свойством `constructor`. Свойство `constructor` указывает на саму функцию. Рассмотрим следующий пример для лучшего понимания этой модели:

```
function Student()
{
  this.name = "Eden";
}
var obj = new Student();
console.log(obj.__proto__.constructor == Student); // Выведет "true"
console.log(obj.__proto__ == Student.prototype); // Выведет "true"
```

Чтобы добавить новые методы в экземпляры конструктора, их следует добавлять в свойство `prototype` конструктора, как мы делали это раньше. Не следует добавлять методы, используя ключевое слово `this` в теле конструктора, потому что в этом случае каждый экземпляр конструктора получит свои копию методов, а это ведет к неэффективному расходованию памяти. Когда методы подключаются к свойству `prototype` конструктора, создается только одна копия каждой функции, которую совместно будут использовать все экземпляры. Для лучшего понимания этого рассмотрим следующий пример:

```
function Student(name)
{
  this.name = name;
}
Student.prototype.printName = function(){
  console.log(this.name);
}
var s1 = new Student("Eden");
var s2 = new Student("John");
function School(name)
{
  this.name = name;
  this.printName = function(){
    console.log(this.name);
  }
}
var s3 = new School("ABC");
var s4 = new School("XYZ");
console.log(s1.printName == s2.printName);
console.log(s3.printName == s4.printName);
```

Результат выполнения кода:

```
true
false
```

Здесь объекты `s1` и `s2` совместно используют одну и ту же функцию `printName`, что ведет к экономии памяти, в то время как объекты `s3` и `s4` содержат две разные функции с именем `printName`, что увеличивает потребление памяти. В этом нет никакой необходимости, так как обе эти функции делают одно и то же. Следовательно, добавлять методы экземпляров нужно в свойство `prototype` конструктора.

Реализация иерархии наследования в конструкторах не так проста, как для литералов объектов. Поскольку конструктору дочернего объекта нужно вызывать конструктор родительского объекта, содержащего логику инициализации родительского объекта, мы должны добавить методы свойства `prototype` родительского конструктора в свойство `prototype` дочернего конструктора, чтобы использовать их в дочернем конструкторе. Для этого не существует предопределенного способа. Разработчики на JavaScript изобретают для этого собственные способы. Я приведу вам самый распространенный из них.

Следующий пример демонстрирует реализацию наследования при создании объектов с помощью конструкторов:

```
function School(schoolName)
{
  this.schoolName = schoolName;
}
School.prototype.printSchoolName = function(){
  console.log(this.schoolName);
}
function Student(studentName, schoolName)
{
  this.studentName = studentName;
  School.call(this, schoolName);
}
Student.prototype = new School();
Student.prototype.printStudentName = function(){
  console.log(this.studentName);
}
var s = new Student("Eden", "ABC School");
s.printStudentName();
s.printSchoolName();
```

Результат выполнения:

```
Eden
ABC School
```

Здесь родительский конструктор вызывается с помощью метода `call` объекта функции. Чтобы унаследовать методы, мы создали эк-

земпляр родительского конструктора и присвоили его свойству `prototype` дочернего конструктора.

Это не лучший способ реализации наследования в конструкторах, так как в нем есть много потенциальных проблем. Например, если родительский конструктор делает что-то еще, кроме инициализации свойств, например, выполняет операции с DOM, присваивание нового экземпляра родительского конструктора свойству `prototype` дочернего конструктора может вызвать проблемы.

Классы в спецификации ES6 обеспечивают более простой и удобный способ наследования существующих конструкторов и классов. Мы рассмотрим эти способы ниже в этой главе.

Конструкторы элементарных типов данных

Элементарные типы данных, такие как `Boolean`, `String` и `Number`, имеют свои аналоги конструкторов. Их неявные конструкторы действуют подобно оберткам вокруг этих элементарных типов. Например, конструктор `String` создает объекты строк с внутренним свойством `[[PrimitiveValue]]`, фактически содержащим элементарное значение.

Во время выполнения, где это необходимо, элементарные значения, обернутые в аналоги конструкторов, а также соответствующие объекты, интерпретируются как элементарные значения, поэтому код работает правильно. Рассмотрим следующий пример, чтобы понять, как это работает:

```
var s1 = "String";
var s2 = new String("String");
console.log(typeof s1);
console.log(typeof s2);
console.log(s1 == s2);
console.log(s1.length);
```

Результат выполнения:

```
string object
true
6
```

Здесь переменная `s1` содержит значение элементарного типа, а переменная `s2` является объектом, несмотря на то, что оператор == возвращает результат `true`. Переменная `s1` имеет элементарный тип, но

при обращении к ее свойству `length` мы получили нужный результат, а ведь элементарные типы не должны иметь свойств. Все это происхо-

Определение классов

Существует два способа определения функций: в виде объявлений и выражений. Аналогично существует два способа определения классов: в виде объявлений и выражений.

Объявление класса

Чтобы определить класс с помощью объявления, следует использовать ключевое слово `class` и имя класса.

Следующий пример демонстрирует определение класса с помощью объявления:

```
class Student
{
  constructor(name)
  {
    this.name = name;
  }
}
var s1 = new Student("Eden");
console.log(s1.name); // Выведет "Eden"
```

Здесь объявляется класс `Student`. В объявлении определяется метод `constructor`, который создает экземпляр класса – объект – и присваивает значение свойству `name` объекта.

Тело класса заключено в фигурные скобки `{}`. В нем мы должны определить методы. Методы определяются без ключевого слова `function` и не разделяются запятыми.

Классы интерпретируются как функции, внутреннее имя класса интерпретируется как имя функции, а тело метода `constructor` рассматривается как тело функции.

В классе может быт только один метод `constructor`. Попытка определить более одного конструктора приведет к исключению `SyntaxError`.

Весь код внутри тела класса должен по умолчанию соответствовать режиму `strict`.

Тот же код можно записать в виде функции:

```
function Student(name)
{
  this.name = name;
}
var s1 = new Student("Eden");
console.log(s1.name); // Выведет "Eden"
```

Чтобы доказать, что класс является функцией, рассмотрим следующий код:

```
class Student
{
  constructor(name)
  {
    this.name = name;
  }
}
function School(name)
{
  this.name = name;
}
console.log(typeof Student);
console.log(typeof School == typeof Student);
```

Результат выполнения кода:

```
function
true
```

Из примера видно, что класс является функцией. Это просто новый синтаксис создания функции.

Выражение класса

Выражение класса имеет синтаксис, похожий на синтаксис объявления класса. Тем не менее, в выражении класса можно опустить имя класса. Тело класса и его поведение одинаковы для обоих способов.

Следующий пример демонстрирует определение класса с помощью выражения:

```
var Student = class {
  constructor(name)
  {
    this.name = name;
  }
}
var s1 = new Student("Eden");
console.log(s1.name); // Выведет "Eden"
```

Здесь мы сохранили ссылку на класс в переменной, а затем использовали ее для создания объектов.

Тот же код можно записать в виде функции:

```
var Student = function(name) {
  this.name = name;
```

```
}
var s1 = new Student("Eden");
console.log(s1.name); // Выведет "Eden"
```

Методы прототипа

Все методы в теле класса добавляются в свойство `prototype` класса. Свойство `prototype` является прототипом объектов, созданных с помощью класса.

Следующий пример демонстрирует добавление методов к свойству `prototype` класса:

```
class Person
{
  constructor(name, age)
  {
    this.name = name;
    this.age = age;
  }
  printProfile()
  {
    console.log("Name is: " + this.name + " and Age is: " + this.age);
  }
}
var p = new Person("Eden", 12)
p.printProfile();
console.log("printProfile" in p.__proto__);
console.log("printProfile" in Person.prototype);
```

Результат выполнения:

```
Name is: Eden and Age is: 12
true
true
```

Здесь можно видеть, что метод `printProfile` был добавлен в свойство `prototype` класса.

Тот же код можно записать в виде функции:

```
function Person(name, age)
{
  this.name = name;
  this.age = age;
}
Person.prototype.printProfile = function()
{
  console.log("Name is: " + this.name + " and Age is: " + this.age);
}
var p = new Person("Eden", 12)
```

```
p.printProfile();
console.log("printProfile" in p.__proto__);
console.log("printProfile" in Person.prototype);
```

Результат выполнения:

```
Name is: Eden and Age is: 12
true
true
```

Методы get и set

В спецификации ES5 для добавления свойства со средством доступа мы должны были использовать метод `Object.defineProperty()`. Спецификация ES6 вводит префиксы `get` и `set` для методов. Эти методы можно добавлять в литералы объектов или в классы для определения атрибутов свойств со средствами доступа.

Когда методы определяются в теле класса, они будут добавлены к свойству `prototype` класса.

Следующий пример демонстрирует определение методов `get` и `set` в классе:

```
class Person
{
  constructor(name)
  {
    this._name_ = name;
  }
  get name(){
    return this._name_;
  }
  set name(name){
    this._name_ = name;
  }
}
var p = new Person("Eden");
console.log(p.name);
p.name = "John";
console.log(p.name);
console.log("name" in p.__proto__);
console.log("name" in Person.prototype);
console.log(Object.getOwnPropertyDescriptor(p.__proto__, "name").set);
console.log(Object.getOwnPropertyDescriptor(Person.prototype, "name").get);
console.log(Object.getOwnPropertyDescriptor(p, "_name_").value);
```

Результат выполнения:

```
Eden
John
```

```
true
true
function name(name) { this._name_ = name; }
function name() { return this._name_; }
John
```

В примере создано свойство со средствами доступа для сокрытия свойства `_name_`. Мы также вывели некоторую информацию, чтобы доказать, что свойство `name` является свойством со средствами доступа и добавляется к свойству `prototype` класса.

Метод генератора

Чтобы метод класса или литерала объекта мог действовать как генератор, к нему следует добавить префикс `*`.

Метод генератора добавляется к свойству `prototype` класса.

Следующий пример демонстрирует определение метода генератора в классе:

```
class myClass
{
  * generator_function()
  {
    yield 1;
    yield 2;
    yield 3;
    yield 4;
    yield 5;
  }
}
var obj = new myClass();
let generator = obj.generator_function();
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().value);
console.log(generator.next().done);
console.log("generator_function" in myClass.prototype);
```

Результат выполнения кода:

```
1
2
3
4
5
true
true
```

Статические методы

Методы, определяемые в теле класса с префиксом `static`, называются статическими методами. Статические методы являются собственными методами класса, то есть, они добавляются не к свойству `prototype` класса, а в сам класс. Например, метод `String.fromCharCode()` является статическим методом конструктора `String`, то есть `fromCharCode` является собственностью самой функции `String`.

Статические методы часто используются для создания вспомогательных функций.

Следующий пример демонстрирует определение и использование статического метода класса:

```
class Student
{
  constructor(name)
  {
    this.name = name;
  }
  static findName(student)
  {
    return student.name;
  }
}
var s = new Student("Eden");
var name = Student.findName(s);
console.log(name); // Выведет "Eden"
```

Здесь метод `findName` является статическим методом класса `Student`. Тот же код можно записать в виде функции:

```
function Student(name)
{
  this.name = name;
}
Student.findName = function(student){
  return student.name;
}
var s = new Student("Eden");
var name = Student.findName(s);
console.log(name); //Output "Eden"
```

Реализация наследования классов

Ранее в этой главе мы видели, как трудно реализовать иерархию наследования с помощью функций. Спецификация ES6 стремится упростить эту работу, вводя предложение `extends` и ключевое слово

super для классов.

При использовании предложения extends класс может наследовать статические и не статические свойства другого конструктора (которые могут быть или не быть определены с помощью класса).

Ключевое слово super имеет двойное назначение:

- в методе constructor класса для вызова родительского конструктора;
- внутри методов класса для ссылок на статические и не статические методы родительского конструктора.

Следующий пример демонстрирует реализацию иерархии наследования в конструкторах с помощью предложения extends и ключевого слова super:

```
function A(a)
{
  this.a = a;
}
A.prototype.printA = function(){
  console.log(this.a);
}
class B extends A
{
  constructor(a, b)
  {
    super(a);
    this.b = b;
  }
  printB()
  {
    console.log(this.b);
  }
  static sayHello()
  {
    console.log("Hello");
  }
}
class C extends B
{
  constructor(a, b, c)
  {
    super(a, b);
    this.c = c;
  }
  printC()
  {
    console.log(this.c);
```

```
    }
    printAll()
    {
      this.printC();
      super.printB();
      super.printA();
    }
  }
  var obj = new C(1, 2, 3);
  obj.printAll();
  C.sayHello();
```

Результат выполнения:

```
3
2
1
Hello
```

Здесь `A` – это конструктор функции, класс `B` – это класс, наследующий класс `A`, класс `C` – это класс, наследующий класс `B`, а так как класс `B` наследует класс `A`, класс `C` так же наследует класса `A`.

Та

```
class myClass
{
  static ["my" + "Method"](){
    console.log("Hello");
  }
}
myClass["my" + "Method"]() ; // Выведет "Hello"
```

Вычисляемые имена свойств также позволяют использовать символы в качестве ключей методов. Следующий пример демонстрирует это:

```
var s = Symbol("Sample");
class myClass
{
  static [s]()
  {
    console.log("Hello");
  }
}
myClass[s]() ; // Выведет "Hello"
```

Атрибуты свойств

При использовании классов, атрибуты статических и нестатических свойств конструктора отличаются от объявленных с помощью функции:

- статические методы доступны для записи и настройки, но недоступны для перечисления;
- свойства `prototype` и `prototype.constructor` класса недоступны для записи, перечисления и настройки;
- свойство `prototype` доступно для записи и настройки, но недоступно для перечисления.

Классы не всплывают!

```
  static get [Symbol.species]()
  {
    return Array;
  }
}
class myCustomArray2 extends Array{}
var arr1 = new myCustomArray1(0, 1, 2, 3, 4);
var arr2 = new myCustomArray2(0, 1, 2, 3, 4);
console.log(arr1 instanceof myCustomArray1);
console.log(arr2 instanceof myCustomArray2);
arr1 = arr1.map(function(value){ return value + 1; })
arr2 = arr2.map(function(value){ return value + 1; })
console.log(arr1 instanceof myCustomArray1);
console.log(arr2 instanceof myCustomArray2);
console.log(arr1 instanceof Array);
console.log(arr2 instanceof Array);
```

Результат выполнения:

```
true
true
false
true
true
false
```

При создании библиотек JavaScript рекомендуется в методах конструкторов в вашей библиотеке всегда обращали внимание на свойство `@@species`, прежде чем вернуть новый экземпляр. Следующий пример демонстрирует это:

```
// Допустим, что myArray1 является частью библиотеки
class myArray1
{
  // Свойство @@species по умолчанию. Будет наследоваться
  // дочерними классами
  static get [Symbol.species]()
  {
    // конструктор по умолчанию
    return this;
  }
  mapping()
  {
    return new this.constructor[Symbol.species]();
  }
}
class myArray2 extends myArray1
{
  static get [Symbol.species]()
  {
```

```
    return myArray1;
  }
}
var arr = new myArray2();
console.log(arr instanceof myArray2); // Выведет "true"
arr = arr.mapping();
console.log(arr instanceof myArray1); // Выведет "true"
```

Если для вас нежелательно определять свойство по умолчанию `@@species` в родительских конструкторах, можно использовать условие `if...else` для проверки присутствия свойства `@@species`. Но предыдущая модель предпочтительнее. Ее так же использует встроенный метод `map()`.

Все встроенные методы конструкторов JavaScript в спецификации ES6 учитывают свойство `@@species`, прежде чем вернуть новый экземпляр. Например, методы `Array`, `Map`, `ArrayBuffer`, `Promise` и другие конструкторы проверяют свойство `@@species`, если они возвращают новый экземпляр.

Неявный параметр new.target

Спецификация ES6 добавляет во все функции параметр `new.target`. Точка является частью имени параметра.

По умолчанию параметр `new.target` получает значение `undefined`. При вызове функции в качестве конструктора значение параметра `new.target` определяется следующими условиями:

- если конструктор вызывается с помощью оператора `new`, `new.target` указывает на конструктор;
- если конструктор вызывается с помощью ключевого слова `super`, `new.target` в нем имеет такое же значение, как `new.target` в конструкторе, который вызывает `super`.

Внутри стрелочной функции `new.target` имеет такое же значение, как `new.target` в окружающей ее не стрелочной функции.

Следующий пример демонстрирует это:

```
function myConstructor()
{
  console.log(new.target.name);
}
class myClass extends myConstructor
{
  constructor()
  {
```

```
      super();
    }
  }
  var obj1 = new myClass();
  var obj2 = new myConstructor();
```

Результат выполнения:

```
myClass
myConstructor
```

Использование super в литералах объектов

Ключевое слово `super` также может использоваться в коротких методах литералов объектов. Ключевое слово `super` в таких методах имеет то же значение, что и свойство `[[prototype]]` литерала.

В литералах объектов, ключевое свойство `super` используется для доступа к переопределяющим свойствам в дочерних объектах.

Следующий пример демонстрирует использование ключевого свойства `super` для литералов объектов:

```
var obj1 = {
  print(){
    console.log("Hello");
  }
}
var obj2 = {
  print(){
  super.print();
  }
}
Object.setPrototypeOf(obj2, obj1);
obj2.print(); // Выведет "Hello"
```

ГЛАВА 8. Модульное программирование

Модульное программирование является одним из наиболее важных и часто используемых методов организации программ. К сожалению, JavaScript не имеет встроенной поддержки модулей, поэтому программисты на JavaScript часто используют альтернативные методы модульного программирования. Но теперь спецификация ES6 официально ввела модули в JavaScript.

Это глава посвящена созданию и импортированию модулей в JavaScript. Здесь вы сначала узнаете, как создавались модули раньше, а потом перейдете к изучению новой системы модулей, которая была введена в спецификации ES6 и известна как «модули ES6».

В этой главе мы рассмотрим:

- общие понятия модульного программирования;
- преимущества модульного программирования;
- основы модулей IIFE, AMD, UMD и CommonJS;
- создание и импортирование модулей ES6;
- основы загрузчика модулей;
- создание базовой библиотеки JavaScript с помощью модулей.

Введение в модули JavaScript

Практика разделения программ и библиотек на модули называется модульным программированием.

В JavaScript модуль представляет коллекцию связанных объектов, функций и других компонентов программы или библиотеки, хранящихся вместе и изолированных от остальной части программы или библиотеки.

Модуль экспортирует некоторые переменные, чтобы дать внешней программе доступ к компонентам, заключенным в модуле. Чтобы ис-

пользовать модуль, программа должна импортировать его и переменные, экспортируемые модулем.

Модуль также можно разделить на меньшие модули, которые часто называют подмодулями, и создать тем самым иерархию модулей.

Модульное программирование имеет много преимуществ. Некоторые из них:

- код получается более простым и лучше организованным, если разделить его на несколько модулей;
- модульное программирование помогает уменьшить количество глобальных переменных, то есть, устраняет проблему глобальных переменных, так как модули взаимодействуют не с помощью глобальной области, и каждый из них имеет свою область видимости;
- упрощает повторное использование кода, так как импорт модулей в различные проекты выполняется несложно;
- позволяет нескольким программистам совместно работать над одной программой или библиотекой, которые могут поделить код на модули с конкретной функциональностью;
- ошибки приложения легче выявлять, локализовав их в определенном модуле.

Реализация модулей по-старому

До появления спецификации ES6, язык JavaScript не поддерживал модули. Для реализации модулей разработчики пользовались другими методами или сторонними библиотеками.

В спецификации ES5 обычно используются следующие способы реализации модулей: **немедленно вызываемые функции-выражения** (Immediately-invoked function expression, IIFE), **асинхронное определение модулей** (Asynchronous Module Definition, AMD), **CommonJS** и **универсальное определение модулей** (Universal Module Definition, UMD). Поскольку эти способы не были встроены в JavaScript, они имели несколько проблем. Давайте сделаем обзор этих старых способов реализации модулей.

Немедленно вызываемые функции-выражения

Немедленно вызываемые функции-выражения (IIFE) используются для создания анонимных функций, которые вызывают себя сами.

Создание модулей с помощью приема IIFE является самым популярным способом создания модулей.

Рассмотрим на примере создание такого модуля:

```
// Начало модуля
(function(window){
  var sum = function(x, y){
    return x + y;
  }
  var sub = function(x, y){
    return x - y;
  }
  var math = {
    findSum: function(a, b){
      return sum(a,b);
    },
    findSub: function(a, b){
      return sub(a, b);
    }
  }
  window.math = math;
})(window)
// Конец модуля
console.log(math.findSum(1, 2)); // Выведет "3"
console.log(math.findSub(1, 2)); // Выведет "-1"
```

Здесь модуль создается с помощью приема IIFE. Переменные `sum` и `sub` являются глобальными для модуля, но не видны за его пределами. Переменная `math` экспортируется модулем в основную программу для предоставления своей функциональности.

Этот модуль работает независимо от программы и может импортироваться в любую другую программу простым копированием его исходного кода или импортированием его, как отдельного файла.

Библиотеки, использующие IIFE, такие как JQuery, обертывают все программные интерфейсы IIFE в один модуль IIFE. Когда программа использует библиотеку JQuery, она автоматически импортирует модуль.

Асинхронное определение модулей

Асинхронное определение модулей (AMD) является спецификацией реализации модулей в браузере. Она разработана с учетом ограничений браузеров, то есть, импортирует модули асинхронно, предотвращая блокирование загрузки веб-страницы. Так как спецификация

AMD не имеет встроенной поддержки в браузерах, необходимо использовать специальную библиотеку AMD. Библиотека RequireJS является наиболее популярной библиотекой AMD.

Рассмотрим пример создания и импортирования модулей с помощью RequireJS. В соответствии со спецификацией AMD, каждый модуль должен храниться в отдельном файле. Итак, сначала создадим файл с именем `math.js`, представляющий модуль. Следующий фрагмент кода будет помещен в модуль:

```
define(function(){
  var sum = function(x, y){
    return x + y;
  }
  var sub = function(x, y){
    return x - y;
  }
  var math = {
    findSum: function(a, b){
      return sum(a,b);
    },
    findSub: function(a, b){
      return sub(a, b);
    }
  }
  return math;
});
```

Здесь модуль экспортирует переменную `math` для предоставления своей функциональности.

Теперь создадим файл `index.js`, представляющий основную программу, которая импортирует модуль и его переменные. Следующий код должен быть помещен в файл `index.js`:

```
require(["math"], function(math){
  console.log(math.findSum(1, 2)); // Выведет "3"
  console.log(math.findSub(1, 2)); // Выведет "-1"
})
```

Здесь переменная `math` в первом параметре является представляет имя файла, который интерпретируется как модуль AMD. Расширение `.js` файла добавляется библиотекой RequireJS автоматически.

Переменная `math` во втором параметре ссылается на экспортированную переменную.

Модуль импортируется в асинхронном режиме, и обратный вызов также выполняется асинхронно.

CommonJS

CommonJS – это спецификация реализации модулей в **Node.js**. Согласно спецификации CommonJS, каждый модуль должен храниться в отдельном файле. В CommonJS модули импортируются синхронно.

Рассмотрим пример создания и импортирования модуля с помощью CommonJS. Сначала создадим файл с именем `math.js`, представляющий модуль. Следующий фрагмент окажется внутри модуля:

```
var sum = function(x, y){
  return x + y;
}
var sub = function(x, y){
  return x - y;
}
var math = {
  findSum: function(a, b){
    return sum(a,b);
  },
  findSub: function(a, b){
    return sub(a, b);
  }
}
exports.math = math;
```

Здесь модуль экспортирует переменную `math` для предоставления своей функциональности.

Теперь создадим файл `index.js` с основной программой, импортирующей модуль. Следующий код должен находиться в файле `index.js`:

```
var math = require("./math").math;
console.log(math.findSum(1, 2)); // Выведет "3"
console.log(math.findSub(1, 2)); // Выведет "-1"
```

Здесь переменная `math` представляет имя файла, который интерпретируется как модуль.

Универсальное определение модуля

Мы видели три разных спецификации реализации модулей. Эти три

ваться как модули IIFE, CommonJS или AMD. Поэтому программа может импортировать модули сторонних разработчиков, независимо от того, какая спецификация использовалась.

Наиболее популярным методом UMD является `returnExports`. Технология `returnExports` требует, чтобы каждый модуль хранился в отдельном файле. Итак, давайте создадим файл с именем `math.js`, представляющий модуль. Следующий фрагмент кода будет помещен в модуль:

```
(function (root, factory) {
  // Выяснение окружения
  if (typeof define === 'function' && define.amd) {
    define([], factory);
  } else if (typeof exports === 'object') {
    module.exports = factory();
  } else {
    root.returnExports = factory();
  }
}(this, function () {
  // Определение модуля
  var sum = function(x, y){
    return x + y;
  }
```

ку и, следовательно, могут рассматриваться как стандартные модули JavaScript.

Вы должны стремиться использовать модули ES6 вместо старых способов, потому что они имеют элегантный синтаксис, лучшую производительность. Кроме того, новые библиотеки почти наверняка будут упаковываться в модули ES6.

Давайте подробно рассмотрим модули ES6.

Создание модулей ES6

Каждый модуль ES6 должен храниться в отдельном файле `.js`. Модуль ES6 может содержать любой программный код на JavaScript и экспортировать любое количество переменных.

Модуль может экспортировать переменные, функции, классы и любые другие структуры.

Чтобы экспортировать переменные из модуля, следует использовать оператор `export`. Оператор `export` имеет несколько разных форматов:

```
export {variableName};
export {variableName1, variableName2, variableName3};
export {variableName as myVariableName};
export {variableName1 as myVariableName1, variableName2 as
myVariableName2};
export {variableName as default};
export {variableName as default, variableName1 as myVariableName1,
variableName2};
export default function(){};
export {variableName1, variableName2} from "myAnotherModule";
export * from "myAnotherModule";
```

- седьмой формат работает аналогично пятому, но позволяет использовать выражение вместо имени переменной;
- восьмой формат экспортирует переменные подмодуля;
- девятый формат экспортирует все экспортируемые переменные подмодуля.

Вот некоторые важные моменты, которые вы должны знать об операторе `export`:

- Оператор `export` можно использовать в любом месте в модуле, а не только в конце.
- Модуль может содержать любое количество операторов `export`.
- Нельзя экспортировать переменные по условию. Например, размещение операторов `export` в операторе `if...else` вызовет ошибку. То есть, можно сказать, что структура модуля должна быть статической, так как экспортируемые элементы должны определяться на этапе компиляции.
- Нельзя экспортировать одно и то же имя переменной или псевдоним несколько раз. Но можно экспортировать переменную несколько раз с разными псевдонимами.
- Весь код в модуле выполняется по умолчанию в режиме `strict`.
- Значения экспортируемых переменных могут изменяться только внутри модуля, который их экспортирует.

Импорт модулей в ES6

Чтобы импортировать модуль, необходимо использовать оператор `import`. Оператор `import` имеет различные форматы:

```
import x from "module-relative-path";
import {x} from "module-relative-path";
import {x1 as x2} from "module-relative-path";
import {x1, x2} from "module-relative-path";
import {x1, x2 as x3} from "module-relative-path";
import x, {x1, x2} from "module-relative-path";
import "module-relative-path";
import * as x from "module-relative-path";
import x1, * as x2 from "module-relative-path";
```

Оператор `import` состоит из двух частей: имен импортируемых переменных и относительного пути к модулю.

Ниже перечислены их отличительные особенности:

- Первый формат импортирует псевдоним `default`. Имя `x` является псевдонимом псевдонима `default`.
- Второй формат импортирует переменную `x`.
- Третий формат такой же как второй. Просто `x2` является псевдонимом `x1`.
- Четвертый формат импортирует переменные `x1` и `x2`.
- Пятый формат импортирует переменные `x1` и `x2`. Псевдоним `x3` является псевдонимом переменной `x2`.
- Шестой формат импортирует переменные `x1` и `x2`, и псевдоним `default`. Псевдоним `x` является псевдонимом псевдонима `default`.
- Седьмой формат просто импортирует модуль. Здесь не импортируется ни одна из переменных, экспортируемых модулем.
- Восьмой формат импортирует все переменные, обернув их в объект с именем `x`. Импортируется даже псевдоним `default`.
- Девятый формат аналогичен восьмому. Здесь псевдониму `default` присваивается друго псевдоним.

Вот некоторые важные моменты, которые вы должны знать об операторе `import`:

- Если переменная импортируется под ее псевдонимом, при обращении к ней следует использовать псевдоним, а не фактическое имя, то есть, фактическое имя переменной будет недоступно, доступным будет только псевдоним.
- Оператор `import` импортирует не копии переменных, а сами эти переменные. Поэтому, если переменная изменит значение внутри модуля, изменение станет доступно в импортирующей программе.
- Импортируемые переменные доступны только для чтения, то есть, их нельзя изменить за пределами модуля, который их экспортирует.
- Модуль может быть импортирован только один раз. Если попытаться импортировать его повторно, будет использован ранее импортированный экземпляр.
- Нельзя импортировать модули по условию. Например, размещение оператора `import` в операторе `if...else` вызовет

ошибку. То есть, можно сказать, что импортируемые элементы должны быть определены на этапе компиляции.

- Импорт модулей ES6 выполняется быстрее, чем импорт модулей AMD или CommonJS, так как модули ES6 имеют встроенную поддержку, а также потому, что импорт модулей и экспорт переменных не выполняются по условию, что позволяет реализации JavaScript оптимизировать операцию импортирования.

Загрузчик модулей

Загрузчик модулей является компонентом реализации JavaScript и отвечает за импорт модулей.

Оператор `import` использует встроенный загрузчик.

Встроенные загрузчики в разных реализациях JavaScript используют разные механизмы загрузки модулей. Например, когда модуль импортируется в браузерах, он загружается с сервера. С другой стороны, когда модуль импортируется в Node.js, он загружается из файловой системы.

Загрузка модулей в разных средах также осуществляется поразному для оптимизации производительности. Например, в браузерах, загрузчик модулей загружает и выполняет модули асинхронно, чтобы предотвратить блокировку загрузки веб-страницы.

Поддерживается возможность взаимодействовать со встроенным загрузчиком через его программный интерфейс для настройки его поведения, перехвата загрузки модулей и загрузки модулей по условию.

Программный интерфейс загрузчика также можно использовать для создания собственных загрузчиков.

Спецификация загрузчика модулей не включена в ES6. Это отдельный стандарт, контролируемый группой стандартизации браузеров **WHATWG**. Вы можете найти технические характеристики загрузчика модулей на странице http://whatwg.github.io/loader/.

Спецификация ES6 определяет только операторы `import` и `export`.

Использование модулей в браузерах

Код внутри тега `<script>` не поддерживает оператор `import`, потому что синхронный характер тега несовместим с асинхронностью модулей в браузерах. Вместо него следует использовать новый тег `<module>`.

С помощью тега `<module>` можно определить сценарий как модуль. После этого модуль сможет импортировать другие модули с помощью оператора `import`.

Если потребуется импортировать модуль с помощью тега `<script>`, вы должны будете использовать **программный интерфейс загрузчика модулей**.

Характеристики тега `<module>` не входят в ES6.

Использование модулей в функции eval()

Вы не сможете использовать операторы `import` и `export` в функции `eval()`. Чтобы импортировать модули в функции `eval()`, следует использовать программный интерфейс загрузчика модулей.

Экспорт по умолчанию или экспорт по именам

Когда экспортируется переменная с псевдонимом `default`, это называется **экспортом по умолчанию**. Очевидно, что в модуле может быть только один экспорт по умолчанию, так как псевдоним может использоваться только один раз.

Все остальные виды экспорта, кроме экспорта по умолчанию, называются **экспортом по именам**.

Рекомендуется использовать только один вид экспорта в модуле – по умолчанию или по именам. Совместное их использование нежелательно.

Экспорт по умолчанию используется, когда требуется экспортировать только одну переменную. Соответственно, экспорт по именам используется для экспорта нескольких переменных.

Пример

Давайте создадим базовую библиотеку JavaScript с помощью модулей ES6. Это поможет понять, как использовать операторы `import` и `export`. Также мы узнаем, как импортировать другие модули из модуля.

Далее мы создадим математическую библиотеку с основными логарифмическими и тригонометрическими функциями. Итак, приступим:

- Создайте файл с именем `math.js` и каталог с именем `math_modules`. Внутри каталога `math_modules`, создайте два файла с именами `logarithm.js` и `trigonometry.js`. Здесь файл `math.js` является главным модулем, а файлы `logarithm.js` и `trigonometry.js` – его подмодулями.

- Поместите следующий код в файл `logarithm.js`:

```
var LN2 = Math.LN2;
var N10 = Math.LN10;
function getLN2()
{
  return LN2;
}
function getLN10()
{
 return LN10;
}
export {getLN2, getLN10};
```

Здесь модуль экспортирует функции по их именам.

Желательно, чтобы модули на нижних уровнях в иерархии экспортировали переменные по отдельности, потому что, возможно, программе потребуется только одна экспортируемая библиотека. В этом случае программа может импортировать модуль и определенную функцию непосредственно. Загрузка всех модулей, когда нужен только один модуль, является неудачным решением с точки зрения производительности.

Аналогично, поместите следующий код в файл `trigonometry.js`:

```
var cos = Math.cos;
var sin = Math.sin;
function getSin(value)
{
  return sin(value);
}
function getCos(value)
{
  return cos(value);
}
export {getCos, getSin};
```

В этом примере мы сделали то же самое. Поместите следующий код в файл `math.js`, который будет главным модулем:

```
import * as logarithm from "math_modules/logarithm";
import * as trigonometry from "math_modules/trigonometry";
export default {
  logarithm: logarithm,
  trigonometry: trigonometry
}
```

Код не содержит каких-либо функций библиотеки. Он предназначен для подключения к программе всей библиотеки. Он сначала

импортирует подмодули, а затем экспортирует свои переменные для основной программы.

Если сценарии `logarithm.js` и `trigonometry.js` зависят от других подмодулей, модуль `math.js` не должен импортировать эти подмодули, потому что `logarithm.js` и `trigonometry.js` уже импортировали их.

С помощью следующего кода программа может импортировать полную библиотеку:

```
import math from "math";
console.log(math.trigonometry.getSin(3));
console.log(math.logarithm.getLN2(3));
```

Итоги

В этой главе мы познакомились с модульным программированием и узнали о различных спецификациях модульного программирования. В заключение мы создали простую библиотеку, используя технологии модульного программирования. Теперь, вы сможете разрабатывать приложения JavaScript с помощью модулей ES6

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

C

CommonJS 160

H

HTML5 89

I

Immediately-Invoked Function Expression 161

P

Perl 34
Python 34

R

Ruby 52

W

WHATWG 169

А

алгебраические операции 49
асинхронный код
  написание 90
  обратные вызовы 94
  события 91
атрибуты свойств 115

Б

базовый объект 142
браузеры
  использование модулей 169
буферные массивы 65

В

восьмеричное представление 44
встроенные символы 79
вычисляемые имена свойств 42

Г

генераторы 81

Д

двоичное представление 44
дескриптор свойств 115
Деструктурирующее присваивание 33
деструктурирующее присваивание массивов 34
  вложенные 36
  дополнение 35
  значения по умолчанию 36
  игнорирование значений 35
  как параметр 36
  описание 34
деструктурирующее присваивание объектов 37
дочерние объекты 142

З

запросы AJAX 90

И

интерфейс состояния батарей 89
исполнитель 96

К

классы
  выражение 148

использование 146
наследование 152
объявление 147
определение 147
статические методы 152
ключевое слово const
область видимости 27
описание 27
ссылки на объект 28
ключевое слово super 42
ключевое слово yield 81
ключевое слово yield* 84
кодовая единица 52
коллекции
Map 69
Set 68
WeakMap 70
WeakSet 69
буферные массивы 65
описание 64
тиризированные массивы 67
конструкторы примитивных типов 145
концевые вызовы 86

Л

ловушка 124

М

массивы
метод Array.from() 60
метод Array.of() 61
метод copyWithin() 63
метод entries() 64
метод fill() 61
метод find() 62
метод findIndex() 63
метод keys() 64
метод values() 64
описание 60
метод apply() 30
метод apply(target, thisValue, arguments) 136
метод catch(onRejected) 104
метод codePointAt() 53
метод constructor
переопределение результата 156
метод construct(target, arguments) 136
метод copyWithin() 63
метод defineProperty(object, property, descriptor) 114
метод deleteProperty() 117
метод deleteProperty(target, property) 133
метод enumerate() 118
метод get(object, property, this) 118
метод getOwnPropertyDescriptor 119
метод getPrototypeOf(object) 120
метод has() 121
метод includes(string, index) 54
метод isExtensible(object) 121
метод isFinite(number) 46
метод isInteger(number) 45
метод isNaN(value) 45
метод isSafeInteger(number) 47
метод ownKeys(object) 122
метод preventExtensions(object) 121
метод Promise.all(iterable) 107
метод Promise.race(iterable) 108
метод Promise.reject(value) 107
метод Promise.resolve(value) 106
метод Reflect.apply() 113
метод Reflect.construct() 113
метод set(object, property, value, this) 119
метод setPrototypeOf(object, prototype) 120
метод Symbol.for(string) 78
многострочные строки 59

Н

неформатированные строки 59
нормализация 55

О

обработчик 124
объект Math 49
объект Reflect 112
объект Symbol 78

П

паттерн Promise 109

подобъект 142
программный интерфейс Proxy 124
программный интерфейс веб-криптографии 110
программный интерфейс состояния батареи 109
производный объект 142
прокси 123

Р

расширенные литералы объектов 41
родительский объект 142

С

свойство Number.EPSILON 48
свойство данных 115
свойство со средствами доступа 115
стрелочные функции
  ключевое слово this 39
  описание 39
  отличие от обычных 41
супер объект 142
схема кодировки UTF-8 52
схема кодировки UTF-16 52

Т

теговая функция 58
типы данных 140
тригонометрические операции 49

Ф

функция clz32(number) 50
функция endsWith 55
функция fround(number) 51
функция imul(number1, number2) 50
функция sign(number) 51
функция trunc(number) 51

Ц

целевой объект 124
цикл for...of 85

Э

экспорт по умолчанию 170

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: **тел. +7 (499) 782-38-89**.

Электронный адрес: **books@alians-kniga.ru**.

Нараян Прасти

Введение в ECMAScript 6

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Рагимов Р. И*
Научный редактор *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 10,23.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Данная книга содержит пошаговые инструкции по использованию новых возможностей ECMAScript 6 вместо устаревших трюков и приемов программирования на JavaScript.

Книга начинается с знакомства со всеми встроенными объектами ES6 и описания создания итераторов ES6. Затем она расскажет, как писать асинхронный код с помощью ES6 в обычном стиле синхронного кода.

Далее описывается использование программного интерфейса рефлексии Reflect API для исследования и изменения свойств объектов. Затем рассматривается создание прокси-объектов и их применение для перехвата и изменения поведения операций с объектами.

И, наконец, демонстрируются устаревшие методы модульного программирования, такие как IIFE, CommonJS, AMD и UMD, и сравниваются с модулями ES6, способными значительно увеличить производительность веб-сайтов.

С этой книгой вы:

- исследуете приемы использования нового синтаксиса ES6;
- познакомитесь с новыми особенностями ES6, основанными на прототипах;
- научитесь выполнять код ES6 в устаревших окружениях, не поддерживающих ES6;
- узнаете, как с помощью объектов Promise писать асинхронный код, более простой для чтения и сопровождения;
- освоите создание и использование итераторов, итерируемых объектов и генераторов;
- познакомьтесь с объектно-ориентированным программированием и созданием объектов с помощью классов;
- научитесь создавать прокси-объекты с помощью программного интерфейса Proxy API и пользоваться ими;
- познакомитесь с приемами создания библиотек JavaScript с помощью модулей ES6.

Кому адресована эта книга

Если вы – программист на JavaScript, обладающий базовыми навыками разработки, и хотите освоить новейшие возможности ES6 для совершенствования своих программ, выполняемых на стороне клиента, эта книга для вас.

SCAN IT!
1053391484
в приложении OZON.ru

ДМ
издателі
www.дм
mkpress.com
ns-kniga.ru
c-книга»
ians-kniga.ru

ISBN 978-5-97060-392-5
9 785970 603925 >