Содержание

Предисловие ... 12
Об авторе ... 16
От издательства ... 17

Глава 1. Значения и переменные ... 18
1.1. Запуск JavaScript ... 18
1.2. Типы и оператор typeof ... 20
1.3. Комментарии ... 21
1.4. Объявления переменных ... 22
1.5. Идентификаторы ... 23
1.6. Числа ... 24
1.7. Арифметические операторы ... 25
1.8. Булевы значения ... 27
1.9. null и undefined ... 27
1.10. Строковые литералы ... 28
1.11. Шаблонные литералы ... 30
1.12. Объекты ... 31
1.13. Синтаксис объектного литерала ... 32
1.14. Массивы ... 33
1.15. JSON ... 34
1.16. Деструктуризация ... 35
1.17. Еще о деструктуризации ... 37
1.17.1. Дополнительные сведения о деструктуризации объектов ... 37
1.17.2. Объявление прочих ... 38
1.17.3. Значения по умолчанию ... 38
Упражнения ... 39

Глава 2. Управляющие конструкции ... 40
2.1. Выражения и предложения ... 40
2.2. Вставка точки с запятой ... 41
2.3. Ветвления ... 44
2.4. Булевость ... 46
2.5. Сравнение ... 46
2.6. Смешанное сравнение ... 48
2.7. Логические операторы ... 49
2.8. Предложение switch ... 51
2.9. Циклы while и do ... 51
2.10. Циклы for ... 52

2.10.1. Классический цикл for 52
2.10.2. Цикл for of 53
2.10.3. Цикл for in 54
2.11. Break и continue 55
2.12. Перехват исключений 57
Упражнения 58

Глава 3. Функции и функциональное программирование 60

3.1. Объявление функций 60
3.2. Функции высшего порядка 61
3.3. Функциональные литералы 62
3.4. Стрелочные функции 63
3.5. Функциональная обработка массива 64
3.6. Замыкания 65
3.7. Крепкие объекты 67
3.8. Строгий режим 69
3.9. Проверка типов аргументов 70
3.10. Передача большего или меньшего числа аргументов 71
3.11. Аргументы по умолчанию 72
3.12. Прочие параметры и оператор расширения 73
3.13. Имитация именованных аргументов с помощью деструктуризации 74
3.14. Поднятие 75
3.15. Возбуждение исключений 77
3.16. Перехват исключений 78
3.17. Ветвь finally 79
Упражнения 80

Глава 4. Объектно-ориентированное программирование 83

4.1. Методы 83
4.2. Прототипы 84
4.3. Конструкторы 87
4.4. Синтаксис классов 88
4.5. Акцессоры чтения и записи 89
4.6. Поля экземпляра и закрытые методы 90
4.7. Статические методы и поля 91
4.8. Подклассы 92
4.9. Переопределение методов 94
4.10. Конструирование подкласса 95
4.11. Классовые выражения 95
4.12. Ссылка this 96
Упражнения 99

Глава 5. Числа и даты 102

5.1. Числовые литералы 102
5.2. Форматирование чисел 103
5.3. Разбор чисел 103

5.4. Функции и константы в классе Number 104
5.5. Математические функции и константы 105
5.6. Большие целые 106
5.7. Конструирование дат 107
5.8. Функции и методы класса Date 110
5.9. Форматирование дат 111
Упражнения 111

Глава 6. Строки и регулярные выражения 114
6.1. Преобразование между строками и последовательностями кодовых точек 114
6.2. Подстроки 115
6.3. Прочие методы класса String 116
6.4. Тегированные шаблонные литералы 119
6.5. Простые шаблонные литералы 120
6.6. Регулярные выражения 121
6.7. Литеральные регулярные выражения 124
6.8. Флаги 125
6.9. Регулярные выражения и Юникод 126
6.10. Методы класса RegExp 127
6.11. Группы 128
6.12. Методы класса String для работы с регулярными выражениями 130
6.13. Еще о методе `replace` 132
6.14. Экзотические возможности 133
Упражнения 134

Глава 7. Массивы и коллекции 137
7.1. Конструирование массива 137
7.2. Свойство length и индексные свойства 138
7.3. Удаление и добавление элементов 139
7.4. Прочие методы изменения массива 141
7.5. Порождение элементов 143
7.6. Поиск элементов 144
7.7. Перебор всех элементов 145
7.8. Разреженные массивы 147
7.9. Редукция 148
7.10. Отображения 151
7.11. Множества 153
7.12. Слабые отображения и множества 154
7.13. Типизированные массивы 155
7.14. Буферные массивы 157
Упражнения 158

Глава 8. Интернационализация 161
8.1. Понятие локали 161
8.2. Задание локали 162

8.3. Форматирование чисел ... 164
8.4. Локализация даты и времени ... 166
8.4.1. Форматирование объектов Date ... 166
8.4.2. Диапазоны ... 167
8.4.3. Относительное время ... 167
8.4.4. Форматирование с точностью до отдельных частей ... 168
8.5. Порядок следования ... 168
8.6. Другие методы класса String, чувствительные к локали ... 170
8.7. Правила образования множественного числа и списков ... 171
8.8. Различные средства, относящиеся к локалям ... 173
Упражнения ... 174

Глава 9. Асинхронное программирование ... 176

9.1. Конкурентные задачи в JavaScript ... 176
9.2. Создание обещаний ... 179
9.3. Немедленно улаживаемые обещания ... 181
9.4. Получение результата обещания ... 182
9.5. Сцепление обещаний ... 182
9.6. Обработка отвергнутых обещаний ... 184
9.7. Выполнение нескольких обещаний ... 185
9.8. Гонка нескольких обещаний ... 186
9.9. Асинхронные функции ... 187
9.10. Асинхронно возвращаемые значения ... 189
9.11. Конкурентное ожидание ... 191
9.12. Исключения в асинхронных функциях ... 191
Упражнения ... 192

Глава 10. Модули ... 196

10.1. Понятие модуля ... 196
10.2. Модули в ECMAScript ... 197
10.3. Импорт по умолчанию ... 197
10.4. Именованный импорт ... 198
10.5. Динамический импорт ... 199
10.6. Экспорт ... 200
10.6.1. Именованный экспорт ... 200
10.6.2. Экспорт по умолчанию ... 201
10.6.3. Экспортируемые средства – это переменные ... 202
10.6.4. Реэкспорт ... 202
10.7. Упаковка модулей ... 203
Упражнения ... 204

Глава 11. Метапрограммирование ... 207

11.1. Символы ... 207
11.2. Настройка с помощью символьных свойств ... 208
11.2.1. Настройка метода toString ... 209
11.2.2. Управление преобразованием типов ... 210

11.2.3. Символ Species......210
11.3. Атрибуты свойств......211
11.4. Перечисление свойств......213
11.5. Проверка наличия свойства......215
11.6. Защита объектов......215
11.7. Создание и обновление объектов......216
11.8. Доступ к прототипу и его обновление......216
11.9. Клонирование объектов......217
11.10. Свойства-функции......220
11.11. Привязка аргументов и вызов методов......221
11.12. Прокси......222
11.13. Класс Reflect......224
11.14. Инварианты прокси......226
Упражнения......228

Глава 12. Итераторы и генераторы......232

12.1. Итерируемые значения......232
12.2. Реализация итерируемого объекта......233
12.3. Закрываемые итераторы......235
12.4. Генераторы......236
12.5. Вложенное yield......238
12.6. Генераторы как потребители......240
12.7. Генераторы и асинхронная обработка......241
12.8. Асинхронные генераторы и итераторы......243
Упражнения......246

Глава 13. Введение в TypeScript......249

13.1. Аннотации типов......250
13.2. Запуск TypeScript......251
13.3. Терминология, относящаяся к типам......252
13.4. Примитивные типы......253
13.5. Составные типы......254
13.6. Выведение типа......256
13.7. Подтипы......259
13.7.1. Правило подстановки......259
13.7.2. Факультативные и лишние свойства......261
13.7.3. Вариантность типов массива и объекта......262
13.8. Классы......263
13.8.1. Объявление классов......263
13.8.2. Тип экземпляра класса......264
13.8.3. Статический тип класса......265
13.9. Структурная типизация......266
13.10. Интерфейсы......267
13.11. Индексные свойства......268
13.12. Более сложные параметры функций......269

13.12.1. Факультативные, подразумеваемые по умолчанию и прочие параметры ... 269
13.12.2. Деструктуризация параметров ... 270
13.12.3. Вариантность типа функции ... 271
13.12.4. Перегрузка ... 273
13.13. Обобщенное программирование ... 275
13.13.1. Обобщенные классы и типы ... 275
13.13.2. Обобщенные функции ... 276
13.13.3. Ограничения на типы ... 277
13.13.4. Стирание ... 278
13.13.5. Вариантность обобщенных типов ... 279
13.13.6. Условные типы ... 280
13.13.7. Отображаемые типы ... 281
Упражнения ... 282

Предметный указатель ... 285

Предисловие

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

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

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

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

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

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

Пять золотых правил

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

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

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

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

Пути к познанию

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

Вот эти значки.

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

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

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

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

Краткое содержание книги

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

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

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

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

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

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

Почему я написал эту книгу

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

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

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

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

Поэтому мне пришлось написать ее самому.

Благодарности

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

Кэй Хорстманн
Берлин
Март 2020

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

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

Об авторе

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

От издательства

Отзывы и пожелания

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

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

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

Скачивание исходного кода примеров

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

Список опечаток

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

Нарушение авторских прав

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

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

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

Глава 1

Значения и переменные

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

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

1.1. Запуск JavaScript

Выполнять встречающиеся в этой книге программы можно несколькими способами.

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

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

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

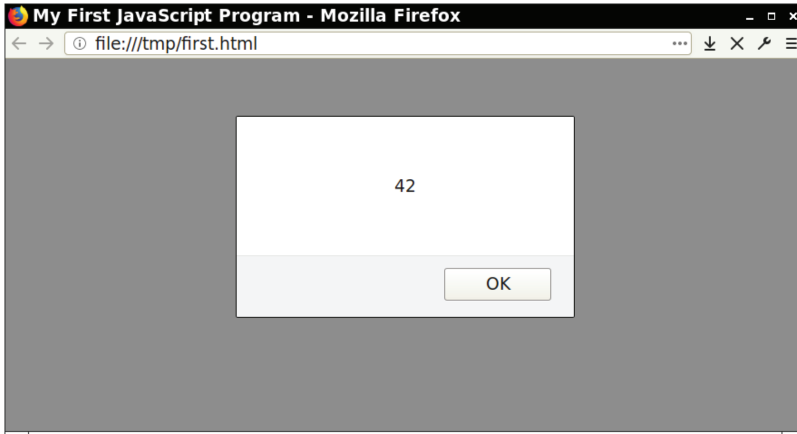

Рис. 1.1 ❖ Выполнение JavaScript-кода в браузере

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

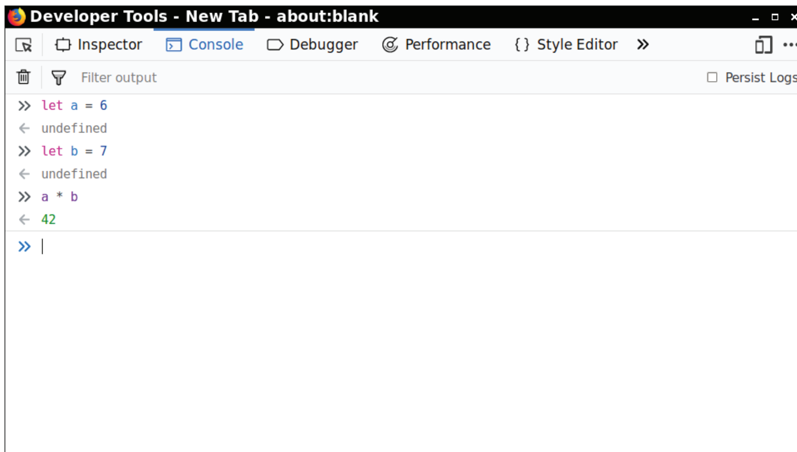

Рис. 1.2 ❖ Выполнение JavaScript-кода на консоли разработчика

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

Рис. 1.3 ❖ Выполнение JavaScript-кода на консоли разработчика

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

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

Затем выполните команду

```
node first.js
```

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

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

1.2. Типы и оператор typeof

Значения в JavaScript могут иметь следующие типы:

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

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

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

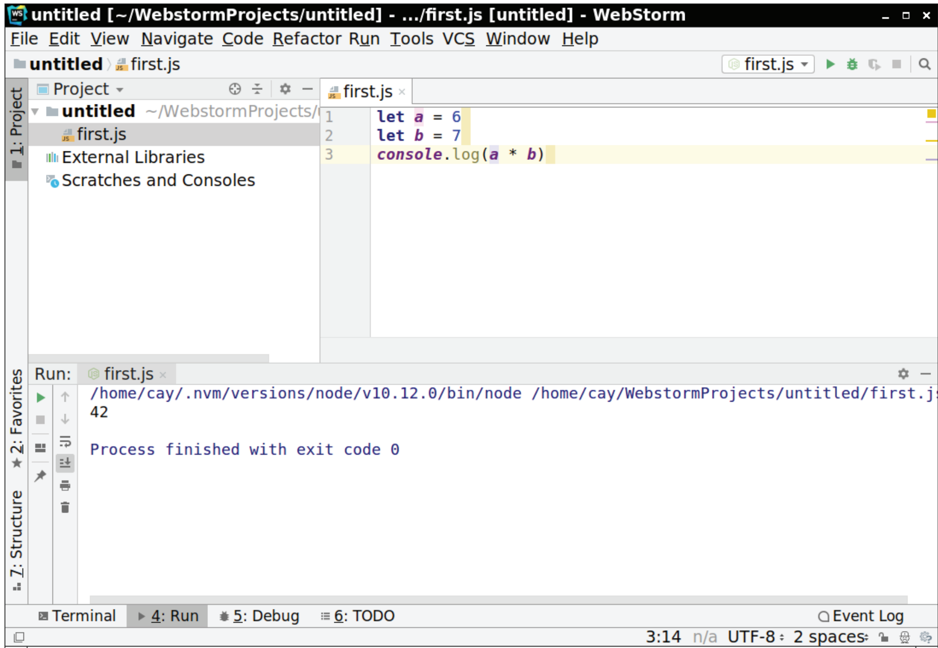

Рис. 1.4 ❖ Выполнение JavaScript-кода в среде разработки

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

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

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

1.3. Комментарии

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

```
// как-то так
```

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

```
/*
  как-то
  так
*/
```

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

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

1.4. Объявления переменных

Для сохранения значения в переменной служит предложение `let`:

```
let counter = 0
```

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

```
counter = 'zero'
```

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

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

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

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

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

```
const PI = 3.141592653589793
```

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

В одном предложении `const` или `let` можно объявить несколько переменных:

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

Но многие программисты предпочитают объявлять каждую переменную в отдельной строке.

> **Предостережение.** Избегайте двух устаревших форм объявления переменных: с помощью ключевого слова `var` и вообще без ключевого слова:
>
> ```
> var counter = 0 // устаревшая форма
> coutner = 1 // обратите внимание на опечатку – будет создана новая переменная!
> ```
>
> У объявления с помощью `var` есть несколько серьезных недостатков, о них будет сказано в главе 3. Создание при первом присваивании, очевидно, опасно. Если сделать опечатку в имени переменной, то будет создана новая переменная. По этой причине такое поведение считается ошибкой в *строгом режиме*, запрещающем устаревшие конструкции. В главе 3 я расскажу, как включить строгий режим.

> **Совет.** В предисловии я перечислил пять золотых правил, следование которым позволит устранить большую часть недоразумений, вызванных «классическими» средствами JavaScript. Первые два из них гласят:
> 1. При объявлении переменных употребляйте ключевые слова `let` или `const`, а не `var`.
> 2. Пользуйтесь строгим режимом.

1.5. Идентификаторы

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

Следующие ключевые слова не разрешается использовать в качестве идентификаторов:

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

В строгом режиме запрещены также такие ключевые слова:

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

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

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

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

1.6. Числа

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

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

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

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

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

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

Метод `toString` преобразует число обратно в строку:

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

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

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

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

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

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

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

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

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

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

1.7. Арифметические операторы

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

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

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

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

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

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

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

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

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

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

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

Например:

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

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

Таблица 1.1 Преобразование в числа и строки

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

Таблица 1.1 (окончание)

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

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

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

1.8. Булевы значения

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

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

1.9. null и undefined

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

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

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

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

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

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

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

1.10. Строковые литералы

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

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

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

Таблица 1.2. Управляющие последовательности для специальных символов

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

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

```
let greeting = 'Hello 🌐'
```

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

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

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

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

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

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

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

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

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

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

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

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

1.11. Шаблонные литералы

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

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

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

```
Hello, WORLD!
```

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

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

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

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

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

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

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

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

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

1.12. Объекты

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

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

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

Разумеется, объект можно сохранить в переменной:

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

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

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

Можно модифицировать существующие свойства и добавлять новые:

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

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

Для удаления свойства служит оператор delete:

```
delete harry.salary
```

Попытка доступа к несуществующему свойству дает undefined:

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

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

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

1.13. Синтаксис объектного литерала

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

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

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

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

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

Для такой ситуации предусмотрен сокращенный синтаксис:

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

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

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

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

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

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

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

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

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

{} – пустой объект, который преобразуется в NaN. После этого отображается результат вычисления (тоже NaN).
На практике такая неоднозначность обычно не возникает. Созданный объектный литерал, как правило, сохраняется в переменной, которая передается в качестве аргумента или возвращается в качестве результата. В этих ситуациях синтаксический анализатор не ожидает блока.
Если вы столкнетесь с ситуацией, в которой объектный литерал ошибочно принят за блок, лекарство простое: заключите литерал в круглые скобки. Пример будет приведен в разделе 1.16 «Деструктуризация».

1.14. Массивы

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

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

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

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

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

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

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

Некоторые элементы массива могут отсутствовать:

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

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

Новые элементы можно добавлять за концом массива:

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

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

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

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

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

```
numbers.lucky = true
```

Это необычный, но вполне допустимый в JavaScript прием.

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

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

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

равно строке `'1,2,3'`.

Массив длины 0 преобразуется в пустую строку.

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

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

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

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

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

1.15. JSON

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

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

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

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

Приведем пример JSON-строки:

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

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

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

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

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

выводит бесполезное сообщение

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

Исправит ситуацию команда `JSON.stringify`:

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

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

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

Или еще проще – сделать их частями объекта:

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

1.16. Деструктуризация

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

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

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

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

Или применить деструктуризацию:

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

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

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

Рассмотрим более сложный случай – сопоставление переменных с элементами массива:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Это то же самое, что

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

или

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

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

1.17. Еще о деструктуризации

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

1.17.1. Дополнительные сведения о деструктуризации объектов

Можно деструктурировать вложенные объекты:

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

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

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

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

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

1.17.2. Объявление прочих

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

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

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

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

Объявление прочих работает и для объектов:

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

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

1.17.3. Значения по умолчанию

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

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

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

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

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

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

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

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

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

Упражнения

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