Дэвид Херман

СИЛА JavaScript

68 способов эффективного использования JS

Addison-Wesley

ПИТЕР

Дэвид Херман

СИЛА JavaScript

68 способов эффективного использования JS

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.988.02-018.1
УДК 004.43
Х39

Херман Д.

Х39 Сила JavaScript. 68 способов эффективного использования JS. — СПб.: Питер, 2013. — 288 с.: ил. — (Серия «Библиотека специалиста»).

ISBN 978-5-496-00524-1

Эта книга поможет вам освоить всю мощь языка программирования JavaScript и научит применять его максимально эффективно. Автор описывает внутреннюю работу языка на понятных практических примерах, которые помогут как начинающим программистам, так и опытным разработчикам повысить уровень понимания JavaScript и существенно обогатить опыт его применения в своей работе.

В книге содержится 68 проверенных подходов для написания «чистого» и работающего кода на JavaScript, которые можно легко использовать на практике. Вы узнаете, как выбирать правильный стиль программирования для каждого конкретного проекта, как управлять непредвиденными проблемами разработки, а также как работать более эффективно на каждом этапе программирования на JavaScript.

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

ББК 32.988.02-018.1
УДК 004.43

ISBN 978-0321812186 англ.

ISBN 978-5-496-00524-1

Содержание

Предисловие 9

Введение 12

JavaScript в сравнении с ECMAScript 13

Веб-программирование. 13

Замечание по поводу параллелизма 14

Благодарности. 15

Об авторе 18

Глава 1. Подготовка к программированию на JavaScript 19

1. Заранее узнайте, какая версия JavaScript используется. . . . 20
2. Разберитесь с числами с плавающей точкой.. 28
3. Остерегайтесь неявного приведения типов данных 32
4. Отдавайте предпочтение примитивам, а не объектным оболочкам 40
5. Избегайте использования оператора == со смешанными типами 42
6. Разберитесь с ограничениями на использование точки с запятой 46
7. Рассматривайте строки в виде последовательности 16-разрядных байтовых представлений кодов. 55

Глава 2. Область видимости переменных 61

8. Старайтесь как можно меньше использовать глобальный объект 61

9. Всегда объявляйте локальные переменные. 65

10. Избегайте ключевого слова with 67

11. Освойте механизм замыканий 72

12. Разберитесь с подъемом переменных 76

13. Используйте немедленно вызываемые функции-выражения для создания локальных областей видимости 79

14. Остерегайтесь использования непереносимых областей видимости, возникающих при использовании именованных функций-выражений 83

15. Остерегайтесь непереносимых областей видимости, возникающих из-за объявления функций внутри локальных блоков 88

16. Избегайте создания локальных переменных с помощью функции eval 91

17. Используйте непрямой вызов функции eval вместо прямого 93

Глава 3. Использование функций 96

18. Разберитесь в различиях между вызовами функций, методов и конструкторов 96

19. Научитесь пользоваться функциями высшего порядка. .. .. .100

20. Используйте метод call для вызова методов с произвольным получателем .. .. .105

21. Используйте метод apply для вызова функций с разным количеством аргументов.. .. .. .107

22. Используйте объект arguments для создания вариативных функций .. .. .110

23. Никогда не вносите изменений в объект arguments . .. .. .112

24. Используйте переменную для сохранения ссылки на объект arguments 115

25. Используйте метод bind для извлечения методов с фиксированным получателем .. .. .117

26. Используйте метод bind для каррирования функций. .. .120

27. При инкапсуляции кода отдавайте предпочтение замыканиям, а не строкам .. .. .122

28. Избегайте использования метода toString функций. .. .. .124
29. Избегайте нестандартных свойств инспектирования стека 127

Глава 4. Объекты и прототипы 130
30. Разберитесь в различиях между механизмами prototype, getPrototypeOf и __proto__ .. .. .131
31. Отдавайте предпочтение функции Object.getPrototypeOf, а не свойству __proto__ 135
32. Никогда не вносите изменения в свойство __proto__. .. .136
33. Создавайте свои конструкторы так, чтобы их не нужно было вызывать с оператором new .. .. .138
34. Храните методы в прототипах .. .. .142
35. Для хранения закрытых данных используйте замыкания. .. .. .144
36. Храните состояние экземпляра только в объектах-экземплярах .. .. .147
37. Разберитесь с неявным связыванием this.. .. .. .150
38. Вызывайте конструкторы суперкласса из конструкторов подкласса 154
39. Не используйте повторно имена свойств суперкласса .. .159
40. Избегайте наследования от стандартных классов.. .. .. .. .161
41. Считайте прототипы деталями реализации .. .. .164
42. Не пытайтесь бездумно вносить изменения методом обезьяньей правки. .. .. .166

Глава 5. Массивы и словари 169
43. Создавайте простые словари только из непосредственных экземпляров объектов. .. .. .169
44. Используйте прототипы равные null для предотвращения прототипного загрязнения .. .. .173
45. Используйте метод hasOwnProperty для защиты от прототипного загрязнения .. .. .176
46. Отдавайте предпочтение массивам, а не словарям при работе с упорядоченными коллекциями .. .. .182
47. Не добавляйте перечисляемые свойства к Object.prototype .. .. .186
48. Избегайте модификации объекта в ходе перечисления 188
49. При последовательном переборе элементов массива отдавайте предпочтение циклу for, а не циклу for...in . .. .194

50. При работе с циклами отдавайте предпочтение итерационным методам 196
51. Повторно используйте обобщенные методы прототипа Array для объектов, похожих на массивы 202
52. Отдавайте предпочтение литералам массивов, а не конструктору Array 205

Глава 6. Разработка библиотек и API 207
53. Придерживайтесь неизменных соглашений 207
54. Рассматривайте вариант undefined как «нет значения». . 210
55. Применяйте для аргументов, требующих описания, объекты параметров 216
56. Избегайте ненужных данных о состоянии 222
57. Используйте структурную типизацию для создания гибких интерфейсов 226
58. Различайте массив и подобие массива 232
59. Избегайте избыточного приведения типов данных 237
60. Выстраивайте цепочки методов. 242

Глава 7. Параллелизм 246
61. Не блокируйте очередь событий при вводе-выводе 247
62. Используйте вложенные или именованные функции обратного вызова для задания последовательности выполнения асинхронных команд 252
63. Не забывайте о существовании игнорируемых ошибок. 257
64. Используйте рекурсию для асинхронных циклов. 262
65. Не блокируйте очередь событий вычислениями. 267
66. Используйте счетчик для выполнения параллельных операций 272
67. Не вызывайте асинхронные функции обратного вызова в синхронном режиме 278
68. Используйте обязательства для более понятной асинхронной логики 282

Глава 1. Подготовка к программированию на JavaScript

Язык JavaScript разрабатывался с расчетом на узнаваемость. Имея синтаксис, напоминающий Java, и конструктивные элементы, общие для многих языков сценариев (такие как функции, массивы, отображения и регулярные выражения), JavaScript создает впечатление языка, быстрое изучение которого под силу любому человеку даже с небольшим опытом программирования. И благодаря небольшому количеству базовых концепций, использующихся в этом языке, новичкам в программировании дается шанс приступить к написанию программ после сравнительно небольшой подготовительной практики.

Однако при всей доступности JavaScript освоение языка занимает куда больше времени и требует глубокого понимания его семантики, характерных особенностей и наиболее эффективных идиом. Каждая глава данной книги описывает разную тематическую область эффективного языка JavaScript, и первая глава начинается с некоторых наиболее фундаментальных тем.

1 ЗАРАНЕЕ УЗНАЙТЕ, КАКАЯ ВЕРСИЯ JAVASCRIPT ИСПОЛЬЗУЕТСЯ

Как и большинство других успешных технологий, JavaScript находится в постоянном развитии. Изначально появившийся на рынке как дополнение к Java для программирования интерактивных веб-страниц, язык JavaScript в конечном счете вытеснил Java из разряда доминирующих языков веб-программирования. Популярность JavaScript привела в 1997 году к формализации этого языка в виде международного стандарта, официально известного как ECMAScript. Сегодня существует множество конкурирующих реализаций JavaScript, соответствующих различным версиям стандарта ECMAScript.

Третье издание стандарта ECMAScript (часто называемое ES3), работа над которым завершилась в 1999 году, остается самой распространенной версией JavaScript. Следующей крупной вехой в стандартизации стала пятая редакция — ES5 (Edition 5), вышедшая в 2009 году. В ES5 было стандартизовано большое количество новых функциональных возможностей, кроме того, стали стандартными некоторые функциональные возможности, ранее широко поддерживаемые, но не имевшие точного определения. Поскольку поддержка ES5 еще не стала повсеместной, те конкретные темы и места в книге, которые касаются ES5, оговариваются особым образом.

Кроме нескольких редакций стандарта, существует также ряд нестандартных функциональных возможностей, поддерживаемых в одних и не поддерживаемых в других реализациях JavaScript. Например, многие JavaScript-движки поддерживают для определения переменных ключевое слово `const`, хотя стандарт ECMAScript не предоставляет каких-либо определений синтаксиса или поведения этого ключевого слова. Более того, механизм функционирования `const` отличается от реализации к реализации. В некоторых случаях переменные, определенные с помощью ключевого слова `const`, защищены от обновления:

```
const PI = 3.141592653589793;
PI = "изменено!";
PI; // 3.141592653589793
```

В то же время в других реализациях `const` просто считается синонимом ключевого слова `var`:

```
const PI = 3.141592653589793;
PI = "изменено!";
PI; // "изменено!"
```

Отслеживание тех или иных функциональных возможностей на разных платформах затруднено как из-за долгой истории JavaScript, так и из-за разнообразия реализаций этого языка. Усугубляет проблему и то, что главная экосистема JavaScript — веб-браузер — не позволяет программистам контролировать доступность той или иной версии JavaScript для выполнения их кода. Поскольку конечными пользователями могут применяться разные версии разных веб-браузеров, веб-программы должны быть написаны с прицелом на единообразную работу во всех браузерах.

В то же время JavaScript в веб-программирования не ограничивается исключительно клиентской стороной. Другие области применения включают программы на стороне сервера, браузерные расширения и создание сценариев для мобильных и настольных систем. В ряде таких случаев вам могут быть доступны куда более конкретные версии JavaScript. Тогда будет иметь смысл воспользоваться дополнительными функциональными возможностями конкретной реализации JavaScript на той или иной платформе.

В данной книге в первую очередь рассматриваются стандартные функциональные возможности JavaScript. Тем не менее важно также упомянуть конкретные широко поддерживаемые, но нестандартные возможности. При работе с новыми стандартизованными или нестандартизованными функциональными возможностями важно разобраться, будут ли ваши приложения работать в средах, поддерживающих такие возможности. В противном случае может оказаться так, что ваши приложения будут нормально работать на вашем собственном компьютере

или в тестируемой инфраструктуре, но откажутся работать при развертывании в других средах. Например, ключевое слово `const` может хорошо работать при тестировании на движке, поддерживающем эту нестандартную функциональную возможность, а потом вызвать сбой с выдачей синтаксической ошибки при развертывании на веб-браузере, не распознающем это ключевое слово.

ES5 предлагает еще один взгляд на версии языка за счет предусмотренного в этом стандарте *строгого режима*. Этот режим позволяет выбрать ограниченную версию JavaScript, в которой отключены некоторые функциональные возможности, которые вызывают наибольшее количество проблем или при работе с которыми чаще всего допускаются ошибки. Синтаксис был разработан с учетом обратной совместимости, поэтому те среды, в которых не реализована проверка строгого режима, способны выполнять код этого режима. Строгий режим включается в программе путем добавления в самом начале программы специальной строковой константы:

```
"use strict";
```

Кроме того, строгий режим может быть включен в функции путем помещения соответствующей директивы в самом начале тела функции:

```
function f(x) {
  "use strict";
  // ...
}
```

Использование в синтаксисе директивы строкового литерала выглядит несколько необычно, но преимущество такого подхода заключается в обратной совместимости: вычисление строкового литерала не имеет побочных эффектов, поэтому движок ES3 выполняет директиву как безопасную инструкцию — он вычисляет строку и тут же игнорирует ее значение. Это позволяет создавать код в строгом режиме, выполняемый на устаревших JavaScript-движках, но с весьма важным ограничением: устаревшие движки не выполняют никаких проверок, присущих стро-

гому режиму. Если не проводить тестирование в среде ES5, то вполне возможно получить код, который не будет работать в среде ES5:

```
function f(x) {
  "use strict";
  var arguments = []; // ошибка: переопределение
                      // переменной arguments
  // ...
}
```

Переопределение переменной `arguments` в строгом режиме запрещено, но в среде, не реализующей проверки строгого режима, этот код будет работать. Развертывание этого кода для постоянной работы вызовет сбой программы в тех средах, где реализован стандарт ES5. Поэтому строгий код непременно нуждается в проверке в среде, полностью совместимой со стандартом ES5.

Подвох при использовании строгого режима заключается в том, что директива `"use strict"` распознается только в начале сценария или функции, что делает ее чувствительной к *объединению сценариев*, когда большие приложения разрабатываются в виде отдельных файлов, а затем объединяются в один файл при развертывании для постоянной работы. Рассмотрим файл, выполнение которого ожидается в строгом режиме:

```
// file1.js
"use strict";
function f() {
  // ...
}
// ...
```

А теперь взглянем на еще один файл, работа которого в строгом режиме не ожидается:

```
// file2.js
// директива строгого режима отсутствует
function g() {
```

```
  var arguments = [];
  // ...
}
// ...
```

Можно ли теперь правильно объединить эти файлы? Если мы начнем с файла file1.js, то весь объединенный файл будет выполняться в строгом режиме:

```
// file1.js
"use strict";
function f() {
  // ...
}
// ...
// file2.js
// директива строгого режима отсутствует
function f() {
  var arguments = []; // ошибка: переопределение
                      // переменной arguments
  // ...
}
// ...
```

А если мы начнем с файла file2.js, то ни одна из частей объединенного файла в строгом режиме выполняться не будет:

```
// file2.js
// директива строгого режима отсутствует
function g() {
  var arguments = [];
  // ...
}
// ...
// file1.js
"use strict";
function f() { // теперь этот код больше
               // не выполняется в строгом режиме
  // ...
}
// ...
```

В собственных проектах вы можете придерживаться политики «только строгий режим» или «только не строгий режим», но если вы хотите создать надежный код, который может быть объединен с любым другим кодом, то для этого есть несколько альтернативных вариантов.

> *Никогда не объединяйте файлы, предназначенные для выполнения в строгом режиме, с файлами, не предназначенными для такой работы.*

Это, наверное, самое простое решение, хотя оно, конечно, ограничивает степень вашего контроля над файловой структурой создаваемого приложения или библиотеки. В лучшем случае вам придется развертывать два отдельных файла, файл, в котором содержатся все «строгие» файлы, и файл, в котором содержатся все «нестрогие» файлы.

> *Объединяйте файлы, заключая их тела в оболочку из немедленно вызываемых функций-выражений (Immediately Invoked Function Expression, IIFE).*

О таких функциях подробно рассказывается в теме 13, а здесь отметим только, что заключение каждого файла в оболочку из функции-выражения позволяет интерпретировать их независимо друг от друга в различных режимах. При этом объединенная версия ранее показанного примера будет иметь следующий вид:

```
// директива строгого режима отсутствует
(function() {
  // file1.js
  "use strict";
  function f() {
    // ...
  }
  // ...
})();
```

```
(function() {
  // file2.js
  // директива строгого режима отсутствует
  function f() {
    var arguments = [];
    // ...
  }
  // ...
})();
```

Поскольку каждый файл помещается в отдельную область видимости, директива строгого режима (или отсутствие таковой) оказывает влияние только на содержимое этого файла. Однако при таком подходе содержимое этих файлов не может считаться интерпретируемым в глобальной области видимости. Например, объявления `var` и `function` не остаются глобальными переменными (более подробно глобальные переменные рассматриваются в теме 8). Этот подход применяется в популярных *модульных системах*, которые управляют файлами и зависимостями, автоматически помещая содержимое каждого модуля в отдельную функцию. Поскольку файлы помещаются в локальные области видимости, в отношении каждого файла может приниматься отдельное решение, касающееся строгого режима.

Пишите свои файлы так, чтобы они вели себя одинаково в любом режиме.

Чтобы написать библиотеку, которая работает в максимально возможном количестве контекстов, вы не должны основываться на предположении, что сценарий объединения обязательно поместит ее в функцию, или предполагать, что код клиента будет работать в строгом или нестрогом режиме. Простейшим способом структурирования своего кода на максимальную совместимость является его написание для выполнения в строгом режиме, но с явным размещением всего кода в функции, которая включает строгий

режим локально. Это похоже на предыдущее решение, в котором содержимое каждого файла помещалось в IIFE, но в данном случае вы создаете IIFE самостоятельно, не полагаясь на то, что это будет сделано за вас инструментальным средством объединения файлов или модульной системой, и явным образом выбираете строгий режим:

```
(function() {
  "use strict";
  function f() {
    // ...
  }
  // ...
})();
```

Заметьте, что этот код считается выполняемым в строгом режиме независимо от того, будет или нет он в дальнейшем объединен со «строгим» или «нестрогим» контекстом. В отличие от этого функция, для которой не выбран строгий режим, все равно будет считаться работающей в строгом режиме, если она при объединении трактуется как код, для которого объявлен строгий режим выполнения. Следовательно, наиболее универсальным вариантом, выбираемым для поддержания совместимости, является написание кода для работы в строгом режиме.

УЗЕЛКИ НА ПАМЯТЬ

- Следует решить, какие версии JavaScript будет поддерживать ваше приложение.
- Нужно убедиться в том, что любые используемые вами функциональные возможности JavaScript поддерживаются всеми средами, в которых будет работать ваше приложение.
- Код, рассчитанный на выполнение в строгом режиме, нужно всегда тестировать в тех средах, в которых выполняется проверка строгого режима.
- Избегайте объединения сценариев, которые по-разному работают со строгим режимом.

2 РАЗБЕРИТЕСЬ С ЧИСЛАМИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Большинство языков программирования имеют несколько типов числовых данных, но JavaScript обходится всего одним типом. Эта особенность отражается на поведении оператора `typeof`, который классифицирует целые числа и числа с плавающей точкой просто как числа:

```
typeof 17;   // "число"
typeof 98.6; // "число"
typeof -2.1; // "число"
```

На самом деле все числа в JavaScript являются числами *с плавающей точкой двойной точности*, то есть 64-разрядными числами, определяемыми стандартом IEEE 754, который широко известен как «doubles» (то есть числа с двойной точностью). Если после этого вас все еще удивляет, откуда берутся целые числа, имейте в виду, что числа с двойной точностью отлично могут представлять целые числа с точностью до 53 разрядов. Все целые числа от –9 007 199 254 740 992 (-2^{53}) до 9 007 199 254 740 992 (2^{53}) являются вполне допустимыми числами с двойной точностью. Так что несмотря на отсутствие отдельного целочисленного типа, JavaScript отлично справляется с целочисленной арифметикой.

Большинство арифметических операторов работает с целыми числами, вещественными числами и с комбинациями тех и других:

```
0.1 * 1.9  // 0.19
-99 + 100; // 1
21 - 12.3; // 8.7
2.5 / 5;   // 0.5
21 % 8;    // 5
```

Однако поразрядные арифметические операторы представляют собой особый случай. Вместо непосредственной работы со своими аргументами как с числами с плавающей точкой, они скрытно преобразуют такие числа в 32-раз-

рядные целые числа. (Точнее, они рассматриваются как 32-разрядные целые числа *с обратным порядком следования байтов и с дополнением до двух*.) Возьмем, к примеру, выражение с поразрядным оператором `OR` (*ИЛИ*):

```
8 | 1; // 9
```

Это вроде бы простое выражение на самом деле требует вычисления в несколько этапов. Как всегда, JavaScript-числа 8 и 1 являются числами с двойной точностью, но они могут также быть представленными как 32-разрядные целые числа, то есть последовательностью, состоящей из тридцати двух единиц и нулей. 32-разрядное целое число 8 имеет следующий вид:

```
00000000000000000000000000001000
```

Вы сами можете в этом убедиться, воспользовавшись применительно к числам методом `toString`:

```
(8).toString(2); // "1000"
```

Аргумент метода `toString` указывает на *основание системы счисления*, в данном случае он обозначает представление по основанию 2 (то есть двоичное число). Из результата отбрасываются лишние нулевые разряды, расположенные слева, поскольку на само значение они не влияют.

Целое число 1 в 32-разрядном представлении имеет следующий вид:

```
00000000000000000000000000000001
```

Поразрядное выражение с оператором `OR` объединяет две последовательности битов, сохраняя любой установленный бит (1), найденный в любом из входящих операндов, в результате получается следующая битовая комбинация:

```
00000000000000000000000000001001
```

Эта последовательность является представлением целого числа 9. Вы можете это проверить с помощью стандарт-

ной библиотечной функции `parseInt`, предоставив ей и в данном случае основание 2:

```
parseInt("1001", 2); // 9
```

(Начальные нулевые биты здесь опять не нужны, поскольку на результат они не влияют.)

Аналогичным образом работают и все остальные поразрядные операторы, переводя вводимые данные в целые числа и выполняя операции над целочисленными битовыми комбинациями перед тем, как преобразовать результаты обратно в стандартные, характерные для JavaScript числа с плавающей точкой. В целом, все эти преобразования требуют от JavaScript-движков дополнительной работы: поскольку числа хранятся в формате с плавающей точкой, они должны быть преобразованы сначала в целые числа, а затем обратно в числа с плавающей точкой. Тем не менее оптимизирующие компиляторы иногда могут понять, когда арифметические выражения и даже переменные работают исключительно с целыми числами, и сохранить данные во внутренних структурах в виде целых чисел, чтобы избежать лишних преобразований.

И последнее предостережение, касающееся чисел с плавающей точкой: если они вам пока еще ничем не досаждали, то, скорее всего, у вас все еще впереди. Числа с плавающей точкой выглядят обманчиво знакомыми, но, к сожалению, они приобрели печальную известность своей неточностью. Даже те арифметические действия, которые выглядят предельно просто, могут выдавать неточные результаты:

```
0.1 + 0.2; // 0.30000000000000004
```

Хотя 64-разрядная точность считается достаточно большой, числа с двойной точностью могут представлять только лишь конечный набор вещественных чисел, в то время как набор таких чисел является бесконечным. Арифметика чисел с плавающей точкой может выдавать только приблизительные результаты, округляя их до ближайшего представимого вещественного числа. Когда

выполняется последовательность вычислений, подобные ошибки округления могут накапливаться, приводя ко все менее и менее точным результатам. Округление также приводит к неожиданным отклонениям от свойств, обычно считающихся арифметическими. Например, вещественные числа считаются *ассоциативными*, а выражается это в том, что для любых вещественных чисел x, y и z всегда соблюдается правило:

$$(x + y) + z = x + (y + z).$$

Однако для чисел с плавающей точкой это правило соблюдается не всегда:

```
(0.1 + 0.2) + 0.3; // 0.6000000000000001
0.1 + (0.2 + 0.3); // 0.6
```

Наличие чисел с плавающей точкой предполагает компромисс между точностью и производительностью. Когда главное — точность, важно быть в курсе ограничений, накладываемых их использованием. Одним из полезных выходов из этой ситуации может стать работа с целыми числами везде, где это только возможно, поскольку они могут быть представлены без округления.

При вычислениях, касающихся денежных единиц, программисты зачастую их масштабируют, чтобы работать с наименьшими валютными единицами. В результате появляется возможность производить вычисления с использованием целых чисел. Например, если считать, что показанные ранее вычисления велись в долларах, то вместо этого мы можем работать с целыми числами в виде центов:

```
(10 + 20) + 30; // 60
10 + (20 + 30); // 60
```

Тем не менее, даже работая с целыми числами, нужно не забывать о том, чтобы все вычисления укладывались в диапазон между -2^{53} и 2^{53}. Зато в этом случае больше не придется переживать за ошибки округления.

УЗЕЛКИ НА ПАМЯТЬ

- В JavaScript используются числа двойной точности с плавающей точкой.
- Целые числа в JavaScript являются не отдельным типом данных, а простым поднабором чисел с двойной точностью.
- В выражениях с поразрядными операторами числа трактуются так, как будто они являются 32-разрядными целыми числами со знаком.
- Не забывайте об ограничениях точности, присущих арифметике чисел с плавающей точкой.

3 ОСТЕРЕГАЙТЕСЬ НЕЯВНОГО ПРИВЕДЕНИЯ ТИПОВ ДАННЫХ

Как ни удивительно, JavaScript может прощать ошибки, связанные с типами данных. Многие языки посчитали бы такое выражение ошибочным:

```
3 + true; // 4
```

Причина в том, что булева величина `true` несовместима с арифметикой. В языках со статической типизацией программа с таким выражением даже не получила бы разрешение на запуск. В некоторых динамически типизированных языках программа бы работала, но подобное выражение вызвало бы исключение. JavaScript не только позволяет программе работать, но и вполне благополучно выдает результат 4!

Немедленное сообщение об ошибке при предоставлении неверного типа данных — довольно редкий случай для JavaScript. Такое, к примеру, может произойти при вызове отсутствующей функцией или при попытке получить свойство объекта `null`:

```
"hello"(1); // ошибка: это не функция
null.x;     // ошибка: невозможно прочитать свойство
            // 'x' объекта null
```

Однако во многих других случаях, вместо того чтобы сообщить об ошибке, JavaScript приводит значение к ожидаемому типу, следуя различным протоколам автоматического преобразования. Например, арифметические операторы `-`, `*`, `/` и `%` пытаются перед проведением вычисления преобразовать свои аргументы в числа. Оператор `+` действует еще более изощренно, поскольку он перегружается либо для сложения чисел, либо для объединения строк в зависимости от типов своих аргументов:

```
2 + 3; // 5
"hello" + " world"; // "hello world"
```

А что произойдет при объединении числа и строки? JavaScript неизменно отдает предпочтение строкам:

```
"2" + 3; // "23"
2 + "3"; // "23"
```

Иногда подобное смешение типов может все сильно запутать, главным образом из-за зависимости от порядка следования операций. Рассмотрим выражение:

```
1 + 2 + "3"; // "33"
```

Поскольку сложение группируется слева (то есть обладает *левой ассоциативностью*), это выражение аналогично следующему:

```
(1 + 2) + "3"; // "33"
```

Рассмотрим другое выражение:

```
1 + "2" + 3; // "123"
```

В отличие от предыдущего результатом этого выражения является строка `"123"` — опять же левая ассоциативность предписывает эквивалентность этого выражения тому выражению, в котором левая операция со знаком плюс взята в скобки:

```
(1 + "2") + 3; // "123"
```

Поразрядные операции приводят к преобразованию не только в числа, но и в тот поднабор чисел, который может быть представлен в виде 32-разрядных целых чисел, о чем уже говорилось в теме 2. Это относится к поразрядным арифметическим операторам (`~`, `&`, `^` и `|`) и к операторам сдвига (`<<`, `>>` и `>>>`).

Описанные варианты приведения типов данных могут быть заманчиво удобными — например, для автоматического преобразования строк, поступающих после пользовательского ввода, из текстового файла или из сетевого потока:

```
"17" * 3; // 51
"8" | "1"; // 9
```

Однако за приведением типов данных могут также скрываться ошибки. Переменная, значение которой окажется равным `null`, не вызовет сбоя при арифметическом вычислении, а будет преобразована в 0; неопределенная переменная будет преобразована в специальное значение с плавающей точкой `NaN`, носящее парадоксальное название «не число (not a number), за что можно упрекнуть разработчиков IEEE-стандарта для чисел с плавающей точкой! Вместо немедленного запуска исключения, это приведение типов данных позволяет вычислению продолжиться зачастую с обескураживающими или непредсказуемыми результатами. Самое плохое, что крайне трудно даже провести тест на значение `NaN`, чему есть две причины.

Во-первых, JavaScript следует за вызывающим недоумение требованием IEEE-стандарта для чисел с плавающей точкой, в соответствии с которым значение `NaN` должно рассматриваться как неравное самому себе. Поэтому проверка значения на равенство `NaN` не работает:

```
var x = NaN;
x === NaN; // false
```

Более того, от стандартной библиотечной функции `isNaN` также мало толку, поскольку она осуществляет собственное неявное приведение типов данных, превращая свои аргу-

менты в числа перед тестированием значения. (Функцию `isNaN`, наверное, следовало бы назвать `coercesToNaN`, то есть приведение типов данных для `NaN`.) Если вы уже знаете, что значение является числом, то можете протестировать его на `NaN` с помощью функции `isNaN`:

```
isNaN(NaN); // true
```

Однако другие значения, которые не имеют никакого отношения к `NaN`, но которые приводимы к `NaN`, неотличимы от `NaN`:

```
isNaN("foo");              // true
isNaN(undefined);           // true
isNaN({});                 // true
isNaN({ valueOf: "foo" }); // true
```

К счастью, для тестирования `NaN` можно применить одну немного странную и непонятную, но одновременно надежную и краткую идиому языка. Поскольку `NaN` является единственным JavaScript-значением, которое считается неравным самому себе, вы всегда можете протестировать переменную на значение `NaN` путем проверки ее на неравенство самой себе:

```
var a = NaN;
a !== a;              // true
var b = "foo";
b !== b;              // false
var c = undefined;
c !== c;              // false
var d = {};
d !== d;              // false
var e = { valueOf: "foo" };
e !== e;              // false
```

Можно также выделить этот программный эталон во вспомогательную функцию с вполне понятным именем:

```
function isReallyNaN(x) {
  return x !== x;
}
```

Однако тестирование значения на неравенство самому себе является настолько лаконичным, что зачастую используется без вспомогательной функции, поэтому важно его видеть и понимать.

Неявные операции приведения типов данных сильно мешают отладке неработающей программы, поскольку за ними скрываются ошибки, причем они затрудняют диагностику этих ошибок. Когда вычисление выполняется неправильно, при отладке лучше всего заняться изучением промежуточных результатов вычисления, производя его в обратном порядке, приближаясь к последней позиции, перед которой что-то пошло не так. Из этой позиции можно проверять аргументы каждой операции, находя те из них, которые имеют неправильный тип. В зависимости от допущенного просчета это может быть логическая ошибка, например использование неправильного арифметического оператора, или ошибка типа данных, например передача неопределенного значения вместо числа.

Объекты также могут быть приведены к примитивам. Наиболее часто так выполняется преобразование в строки:

```
"the Math object: " + Math; // "the Math object:
                            // [object Math]"
"the JSON object: " + JSON; // "the JSON object:
                            // [object JSON]"
```

Объекты преобразуются в строки путем неявного вызова принадлежащего им метода `toString`. Это можно проверить, самостоятельно вызвав следующую функцию:

```
Math.toString(); // "[object Math]"
JSON.toString(); // "[object JSON]"
```

По аналогии с этим объекты могут быть преобразованы в числа, для чего используется принадлежащий им метод `valueOf`. Вы можете управлять преобразованием типов объектов, определяя эти методы:

```
"J" + { toString: function() { return "S"; } }; // "JS"
2 * { valueOf: function() { return 3; } };     // 6
```

И опять все усложняется, когда дело касается перегрузки оператора + либо для объединения строк, либо для сложения чисел. В особенности, когда объект содержит оба метода, и `toString`, и `valueOf`, то совершенно непонятно, какой из них будет вызван оператором +: предполагается, что выбор между объединением и сложением основывается на типах данных, но с учетом неявного приведения типов данных, поскольку на самом деле эти типы не заданы! JavaScript разрешает эту неопределенность, слепо выбирая метод `valueOf`, предпочитая его методу `toString`. Однако это означает, что при намерении выполнить объединение строк с использованием объекта поведение программы может быть неожиданным:

```
var obj = {
  toString: function() {
    return "[object MyObject]";
  },
  valueOf: function() {
    return 17;
  }
};
"object: " + obj; // "object: 17"
```

Мораль этой истории состоит в том, что метод `valueOf` в действительности ориентирован на объекты, представляющие числовые значения, например в отношении объектов `Number`. Для этих объектов методы `toString` и `valueOf` возвращают непротиворечивые результаты — строковое представление или числовое представление одного и того же числа — поэтому перегруженный оператор + всегда ведет себя последовательно независимо от того, для чего используется объект, для объединения или для сложения. В общем, приведение к строковым значениям является гораздо более распространенным и полезным, чем приведение к числовым значениям. Лучше все же избегать метода `valueOf`, если ваш объект не является числовой абстракцией и метод `obj.toString()` не производит строковое представление того, что производит метод `obj.valueOf()`.

Последняя разновидность приведения типов данных известна как *истинность*. Такие операторы, как `if`, `||` и `&&`, сообразуясь с логикой, работают с булевыми значениями, но на самом деле принимают любые значения. В JavaScript значения интерпретируются как булевы в соответствии с простым приведением типов данных. Большинство значений JavaScript являются *истинными*, то есть неявным образом приводятся к значению `true`. Это относится ко всем объектам — в отличие от приведения к строковым или числовым значениям, истинность не вызывает неявным образом какие-либо методы приведения типов данных. Существует всего лишь семь *ложных* значений: `false`, `0`, `-0`, `""`, `NaN` и `undefined`. Все остальные значения истинны. Поскольку числа и строки могут быть ложными, использование истинности для проверки факта определения аргумента функции или свойства объекта не всегда оказывается безопасным. Рассмотрим функцию, которой могут передаваться необязательные аргументы, имеющие значения по умолчанию:

```
function point(x, y) {
  if (!x) {
    x = 320;
  }
  if (!y) {
    y = 240;
  }
  return { x: x, y: y };
}
```

Эта функция игнорирует любые ложные аргументы, в том числе 0:

```
point(0, 0); // { x: 320, y: 240 }
```

Более точный способ проверки факта отсутствия определения заключается в использовании оператора `typeof`:

```
function point(x, y) {
  if (typeof x === "undefined") {
    x = 320;
  }
```

```
  if (typeof y === "undefined") {
    y = 240;
  }
  return { x: x, y: y };
}
```

Эта версия функции `point` правильно отличает `0` от `undefined`:

```
point();     // { x: 320, y: 240 }
point(0, 0); // { x: 0, y: 0 }
```

Еще один подход заключается в сравнении со значением `undefined`:

```
if (x === undefined) { ... }
```

Последствия тестирования на истинность для разработки библиотек и API рассмотрены в теме 54.

УЗЕЛКИ НА ПАМЯТЬ

- Ошибки, связанные с типом данных, могут быть молчаливо спрятаны неявным приведением типов данных.
- Оператор + перегружается для сложения или объединения строк в зависимости от типов данных его аргументов.
- Объекты приводятся к числам посредством метода `valueOf` и к строкам посредством метода `toString`.
- Объекты, имеющие метод `valueOf`, должны иметь реализацию метода `toString`, предоставляющего строковое представление того числа, которое производится методом `valueOf`.
- Для тестирования неопределенных значений нужно использовать оператор `typeof` или выполнять сравнение со значением `undefined`, а не выполнять проверку на истинность.