

# Оглавление

<b>Предисловие</b> .....	<b>9</b>
<b>Введение</b> .....	<b>11</b>
Задача .....	12
О книге. ....	14
Типографские соглашения .....	14
Использование программного кода примеров .....	15
От издательства .....	16
<b>Глава 1. Типы</b> .....	<b>17</b>
Хоть типом назови его, хоть нет.....	18
Встроенные типы .....	19
Значения как типы .....	22
undefined и необъявленные переменные .....	23
typeof для необъявленных переменных .....	24
Итоги .....	28
<b>Глава 2. Значения.</b> .....	<b>30</b>
Массивы .....	30
Подобие массивов .....	32
Строки .....	33
Числа .....	37
Синтаксис работы с числами .....	37
Малые дробные значения .....	42
Безопасные целочисленные диапазоны .....	44

---

Проверка целых чисел . . . . .	45
32-разрядные целые числа (со знаком) . . . . .	46
Специальные значения . . . . .	46
Пустые значения . . . . .	47
Undefined. . . . .	47
Специальные числа . . . . .	50
Специальное равенство . . . . .	57
Значения и ссылки . . . . .	58
Итоги . . . . .	64
<b>Глава 3. Встроенные объекты (natives) . . . . .</b>	<b>66</b>
Внутреннее свойство [[Class]] . . . . .	68
Упаковка. . . . .	69
Ловушки при работе с объектными обертками . . . . .	70
Распаковка . . . . .	71
Встроенные объекты как конструкторы . . . . .	72
Array(..) . . . . .	72
Object(..), Function(..) и RegExp(..) . . . . .	77
Date(..) и Error(..). . . . .	79
Symbol(..). . . . .	81
Встроенные прототипы . . . . .	82
Итоги . . . . .	86
<b>Глава 4. Преобразование типов . . . . .</b>	<b>87</b>
Преобразование значений. . . . .	87
Абстрактные операции . . . . .	90
ToString . . . . .	90
ToNumber. . . . .	97
ToBoolean . . . . .	99
Явное преобразование типов . . . . .	104
Явные преобразования: String <--> Number . . . . .	105

---

Явные преобразования: разбор числовых строк . . . . .	115
Явные преобразования: * --> Boolean . . . . .	120
Неявное преобразование . . . . .	122
Неявное упрощение. . . . .	124
Неявные преобразования: String <--> Number . . . . .	125
Неявные преобразования: Boolean --> Number . . . . .	130
Неявные преобразования: * --> Boolean. . . . .	132
Операторы    и &&. . . . .	134
Преобразование символических имен . . . . .	139
Равенство строгое и нестрогое . . . . .	140
Быстродействие проверки равенства . . . . .	141
Абстрактная проверка равенства . . . . .	142
Особые случаи. . . . .	151
Абстрактное относительное сравнение . . . . .	162
Итоги . . . . .	165
<b>Глава 5. Грамматика. . . . .</b>	<b>166</b>
Команды и выражения . . . . .	167
Завершающие значения команд . . . . .	168
Побочные эффекты выражений . . . . .	171
Правила контекста. . . . .	177
Приоритет операторов . . . . .	186
Ускоренная обработка . . . . .	190
Плотное связывание . . . . .	191
Ассоциативность . . . . .	192
Неоднозначности. . . . .	196
Автоматические точки с запятой . . . . .	198
Исправление ошибок . . . . .	200
Ошибки . . . . .	202
Преждевременное использование переменных. . . . .	204
Аргументы функций . . . . .	205

---

try..finally . . . . .	208
switch . . . . .	212
Итоги . . . . .	215
<b>Приложение А. JavaScript в разных средах . . . . .</b>	<b>218</b>
Дополнение В (ECMAScript) . . . . .	218
Web ECMAScript . . . . .	219
Управляющие объекты . . . . .	221
Глобальные переменные DOM . . . . .	222
Встроенные прототипы . . . . .	223
Прокладки совместимости (shims)/полифилы (polyfills) . . . . .	227
<script>ы . . . . .	229
Зарезервированные слова . . . . .	233
Ограничения реализации . . . . .	234
Итоги . . . . .	235
<b>Об авторе . . . . .</b>	<b>236</b>

## Объектные литералы

Прежде всего, это объектный литерал:

```
// предполагается, что функция `bar()` определена
var a = {
  foo: bar()
};
```

Как мы узнаем, что это объектный литерал? Потому что пара фигурных скобок { .. } — это значение, присваиваемое *a*.



Ссылка *a* называется левосторонним значением, потому что она играет роль приемника для присваивания. Пара { .. } называется правосторонним значением, поскольку используется *просто* как значение (в данном случае как источник присваивания).

## Метки

Что произойдет, если убрать из приведенного выше часть `var a =`?

```
// предполагается, что функция `bar()` определена
{
  foo: bar()
}
```

Многие разработчики считают, что пара { .. } — просто автономный объектный литерал, который ничему не присваивается. Это совершенно не так.

Здесь { .. } — обычный программный блок. В JavaScript подобные автономные блоки { .. } выглядят не особо идиоматично (а уж в других языках и подавно), но это абсолютно законная грамматика JS. Блоки бывают особенно полезны в сочетании с объявлениями с блочной областью видимости.

С функциональной точки зрения программный блок `{ .. }` почти идентичен программному блоку, присоединенному к другой команде: циклу `for/while`, условной команде `if` и т. д.

Но если это обычный блок кода, что это за странный синтаксис `foo: bar()` и как он может быть допустимым?

Все происходит из-за малоизвестной (и честно говоря, не рекомендуемой к использованию) возможности JavaScript, так называемых «команд с метками». `foo` — метка для команды `bar()` (без завершающего символа `;` — см. раздел «Автоматические завершители»). Но в чем смысл команды с метками?

Если бы в JavaScript была команда `goto`, теоретически можно было бы использовать команду `goto foo` и передать управление в указанную точку кода. Команды `goto` обычно считаются ужасной идиомой, которая сильно усложняет понимание кода («спагетти-код»), поэтому *очень хорошо*, что в JavaScript нет обобщенной конструкции `goto`.

Однако JS поддерживает ограниченную специальную форму `goto`. При выполнении команд `continue` и `break` может быть указана метка, в этом случае логика программы выполняет «переход» по аналогии с `goto`. Пример:

```
// цикл с меткой `foo`
foo: for (var i=0; i<4; i++) {
  for (var j=0; j<4; j++) {
    // при совпадении переменных циклов продолжить
    // внешний цикл
    if (j == i) {
      // перейти к следующей итерации цикла
      // с меткой `foo`
      continue foo;
    }

    // пропускать нечетные произведения
    if ((j * i) % 2 == 1) {
      // обычное (без метки) продолжение внутреннего цикла
      continue;
    }
  }
}
```

```

    }
    console.log( i, j );
  }
}
// 1 0
// 2 0
// 2 1
// 3 0
// 3 2

```



`continue foo` не означает «перейти к позиции с меткой `foo`, чтобы продолжить». Смысл другой: «продолжить цикл с меткой `foo` и выполнить его следующую итерацию». Таким образом, это не *произвольный* переход `goto`.

Как видите, итерация с нечетным произведением `3 * 1` пропускается, но переход с помеченным циклом также пропустил итерации `1 * 1` и `2 * 2`.

Возможно, чуть более полезная форма цикла перехода в цикле с меткой — это команда `break` \_\_ из внутреннего цикла, когда вы хотите выйти из внешнего цикла. Без `break` с меткой реализация той же логики была бы громоздкой и некрасивой:

```

// цикл с меткой `foo`
foo: for (var i=0; i<4; i++) {
  for (var j=0; j<4; j++) {
    if ((i * j) >= 3) {
      console.log( "stopping!", i, j );
      break foo;
    }
  }
  console.log( i, j );
}
// 0 0
// 0 1
// 0 2
// 0 3

```

```
// 1 0
// 1 1
// 1 2
// stopping! 1 3
```



`break foo` не означает «перейти к позиции с меткой `foo`, чтобы продолжить». Смысл другой: «выйти из цикла/блока с меткой `foo` и продолжить выполнение после него». Не похоже на `goto` в традиционном смысле, да?

Вероятно, альтернативная реализация `break` без метки потребует одной или нескольких функций доступа к переменным в общей области видимости и т. д. Скорее всего, она создаст больше путаницы, чем `break` с меткой, поэтому `break` с меткой, пожалуй, будет лучшим решением.

Метка может создаваться в блоке без цикла, но только `break` может ссылаться на такую метку без цикла. Команда `break ___` с меткой может осуществлять выход из любого блока с меткой, но выполнить `continue ___` с меткой без цикла нельзя, как нельзя и выполнить `break` без метки из блока:

```
// цикл с меткой `bar`
function foo() {
  bar: {
    console.log( "Hello" );
    break bar;
    console.log( "never runs" );
  }
  console.log( "World" );
}

foo();
// Hello
// World
```

Циклы/блоки с метками встречаются крайне редко, и многие разработчики их не одобряют. Лучше избегать их, если это воз-

можно (например, используя вызовы функций вместо переходов из циклов). Наверное, можно придумать несколько особых ситуаций, в которых они могут быть полезными. Если вы собираетесь использовать переход с меткой, обязательно документировайте происходящее в подробных комментариях!

Очень часто встречается мнение, будто JSON является подмножеством JS, так что строка JSON (например, `{"a":42}`), обратите внимание на кавычки вокруг имени свойства, как требует JSON) рассматривается как действительная программа JavaScript. Неправда! Попробуйте ввести `{"a":42}` в консоли JS, и вы получите сообщение об ошибке.

Дело в том, что метки команд не могут заключаться в кавычки, поэтому `"a"` не является действительной меткой, а следовательно, `:` не может следовать после `"a"`. Таким образом, JSON действительно является подмножеством синтаксиса JS, но сам по себе не является действительной грамматикой JS.

Еще одно в высшей степени распространенное заблуждение из той же серии, что если вы загрузите в теге `<script src=...>` файл JS, который содержит только контент JSON (например, полученный при вызове API), данные будут прочитаны как действительный код JavaScript, но будут недоступны программе. Считается, что JSON-P (практика упаковки данных JSON в вызов функции, как в `foo({"a":42})`) решает эту проблему недоступности, передавая значение одной из функций вашей программы.

Неправда! Абсолютно законное значение JSON `{"a":42}` само по себе выдаст ошибку JS, потому что будет интерпретировано как блок с недействительной меткой. С другой стороны, `foo({"a":42})` является действительным кодом JS, потому что `{"a":42}` интерпретируется как объектный литерал, передаваемый `foo(...)`. Таким образом, можно сказать, что *JSON-P превращает JSON в действительную грамматику JS.*

## Блоки

Другая часто упоминаемая проблема JS (связанная с преобразованием типов — см. главу 4):

```
[] + {}; // "[object Object]"
{} + []; // 0
```

Похоже, отсюда следует, что оператор `+` дает разные результаты в зависимости от того, какой операнд является первым, `[]` или `{}`. Ничего подобного!

В первой строке `{}` входит в выражение оператора `+`, а следовательно, интерпретируется как фактическое значение (пустой объект). В главе 4 объясняется, что `[]` преобразуется в `""`, а следовательно, `{}` также преобразуется в строковое значение: `"[object Object]"`.

Но во второй строке `{}` интерпретируется как автономный пустой блок (который не делает ничего). Блоки не обязаны завершаться символом `;`, так что отсутствие завершителя проблем не создает. Наконец, `[]` — выражение, которое явно преобразует (см. главу 4) `[]` в число, отсюда значение `0`.

## Деструктуризация объектов

Начиная с ES6 пары `{ .. }` также могут встретиться при выполнении «деструктурирующего присваивания», а конкретно при деструктуризации объектов. Пример:

```
function getData() {
  // ..
  return {
    a: 42,
    b: "foo"
  };
}

var { a, b } = getData();
console.log( a, b ); // 42 "foo"
```