O'REILLY®

«Превосходное введение в фундаментальные средства JavaScript, о которых вам никогда не расскажут (да и не смогут рассказать) в описаниях инструментариев языка».

— Дэвид Уолш, старший веб-разработчик, Mozilla

КАЙЛ СИМПСОН

ТИПЫ & ГРАММАТИЧЕСКИЕ КОНСТРУКЦИИ

ВЫ НЕ ЗНАЕТЕ JS

ББК 32.988.02-018
УДК 004.738.5
С37

Симпсон К.

С37 {Вы не знаете JS} Типы и грамматические конструкции. — СПб.: Питер, 2019. — 240 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1266-1

Каким бы опытом программирования на JavaScript вы ни обладали, скорее всего, вы не понимаете язык в полной мере. Это лаконичное руководство исследует типы более глубоко, чем все существующие книги: вы узнаете, как работают типы, о проблемах их преобразования и научитесь пользоваться новыми возможностями.

Как и в других книгах серии «Вы не знаете JS», здесь показаны нетривиальные аспекты языка, от которых программисты JavaScript предпочитают держаться подальше (или полагают, что они не существуют). Вооружившись этими знаниями, вы достигнете истинного мастерства JavaScript.

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1491904190 англ.

Authorized Russian translation of the English edition of You Don't Know JS: Types & Grammar (ISBN 9781491904190)

ISBN 978-5-4461-1266-1

Оглавление

1 Типы

Многие разработчики скажут, что в динамическом языке (таком, как JS) вообще нет *типов*. Посмотрим, что на этот счет сказано в спецификации ES5.1[1]:

> «Алгоритмы из этой спецификации работают со значениями, с каждым из которых связан тип. Возможные значения типов точно определены в этой секции. Типы дополнительно классифицируются на типы языка ECMAScript и типы спецификации.
>
> Тип языка ECMAScript соответствует значениям, с которыми программист ECMAScript работает напрямую с использованием языка ECMAScript. Типы языка ECMAScript — undefined, null, boolean, string, number и object».

Возможно, поклонники языков с сильной (статической) типизацией будут возражать против такого использования слова «тип». В этих языках термин «тип» означает гораздо *больше*, чем в JS.

Кто-то скажет, что JS не должен претендовать на наличие «типов». Правильнее было бы называть эти конструкции «тегами» или, скажем, «подтипами».

---

[1] http://www.ecma-international.org/ecma-262/5.1/.

Вот еще! Мы будем использовать следующее приближенное определение (которое, похоже, было заложено в формулировку из спецификации): *тип* представляет собой внутренний встроенный набор характеристик, которые однозначно определяют поведение конкретного значения и отличают его от других значений — как для движка, так и для разработчика.

Иначе говоря, если и движок, и разработчик интерпретируют значение 42 (число) не так, как они интерпретируют значение `"42"` (строка), то эти два значения имеют разные *типы* — `number` и `string` соответственно. Используя 42, вы *хотите* выполнить некую численную операцию — например, математические вычисления. Но при использовании `"42"` вы *намереваетесь* сделать нечто, связанное со строками, вывести данные на страницу и т. д. Эти два значения относятся к разным типам.

Такое определение ни в коем случае не идеально. Тем не менее для нашего обсуждения достаточно и такого. Кроме того, оно соответствует тому, как JS описывает свое поведение.

Хоть типом назови его, хоть нет...

Если не считать расхождений в академических определениях, почему так важно, есть в JavaScript типы или нет?

Правильное понимание каждого типа и его внутреннего поведения абсолютно необходимо для понимания того, как правильно и точно преобразовывать значения к разным типам (см. главу 4). Практически любая когда-либо написанная программа JS должна преобразовывать значения в той или иной форме, поэтому очень важно, чтобы вы делали это ответственно и уверенно.

Если имеется `число` 42, которое вы хотите интерпретировать как `строку` (например, извлечь `"2"` как символ в позиции 1), очевидно, значение сначала необходимо преобразовать из `числа` в `строку`.

На первый взгляд кажется, что это просто.

Однако такие преобразования могут выполняться множеством разных способов. Некоторые из них выражаются явно, логически объяснимы и надежны. Но если вы будете недостаточно внимательны, преобразования могут выполняться очень странным и неожиданным образом.

Пожалуй, путаница с преобразованиями — один из главных источников огорчений для разработчиков JavaScript. Преобразования часто критиковали как настолько *опасные*, что они относились к дефекту проектирования языка, от которого стоит держаться подальше.

Вооружившись полным пониманием типов JavaScript, мы постараемся показать, почему *плохая репутация* преобразований в основном преувеличена и отчасти незаслуженна. Попытаемся поменять вашу точку зрения, чтобы вы увидели мощь и полезность преобразований. Но сначала нужно гораздо лучше разобраться со значениями и типами.

Встроенные типы

JavaScript определяет семь встроенных типов:

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol` — добавлен в ES6!

Все эти типы, за исключением `object`, называются «примитивами».

Оператор `typeof` проверяет тип заданного значения и всегда возвращает одно из семи строковых значений — как ни странно, между ними и семью встроенными типами, перечисленными ранее, нет полностью однозначного соответствия:

```
typeof undefined     === "undefined"; // true
typeof true          === "boolean";   // true
typeof 42            === "number";    // true
typeof "42"          === "string";    // true
typeof { life: 42 }  === "object";    // true

// Добавлен в ES6!
typeof Symbol()      === "symbol";    // true
```

Шесть перечисленных типов имеют значения соответствующего типа и возвращают строковое значение с тем же именем. `Symbol` — новый тип данных, появившийся в ES6; он будет описан в главе 3.

Возможно, вы заметили, что я исключил из этого списка `null`. Это *особенный* тип. Особенный в том смысле, что его поведение с оператором `typeof` ошибочно:

```
typeof null === "object"; // true
```

Было бы удобно (и правильно!), если бы оператор возвращал `"null"`, но эта исходная ошибка в JS существует уже почти два десятилетия и, скорее всего, никогда не будет исправлена. От этого ошибочного поведения зависит столько существующего веб-контента, что «исправление» ошибки только *создаст* еще больше «ошибок» и нарушит работу многих веб-программ.

Если вы хотите проверить значение `null` по типу, вам понадобится составное условие:

```
var a = null;
(!a && typeof a === "object"); // true
```

`Null` — единственное примитивное значение, которое является «*ложным*» (см. главу 4), но при этом возвращает `"object"` при проверке `typeof`.

Какое же седьмое строковое значение может вернуть `typeof`?

```
typeof function a(){ /* .. */ } === "function"; // true
```

Легко решить, что `function` является высокоуровневым встроенным типом в JS, особенно при таком поведении оператора `typeof`. Тем не менее при чтении спецификации вы увидите, что `function` на самом деле ближе к «подтипу» `object`. А именно функция называется там «вызываемым объектом» — то есть объектом с внутренним свойством `[[Call]]`, которое позволяет активизировать его посредством вызова.

Тот факт, что функции в действительности являются объектами, имеет ряд важных следствий. Самое важное — то, что они могут обладать свойствами. Пример:

```
function a(b,c) {
    /* .. */
}
```

У объекта функции есть свойство `length`, в котором хранится количество формальных параметров в объявлении этой функции:

```
a.length; // 2
```

Так как функция была объявлена с двумя формальными именованными параметрами (`b` и `c`), «длина» функции равна 2.

Как насчет массивов? Они встроены в JS — может, им выделен отдельный тип?

```
typeof [1,2,3] === "object"; // true
```

Нет, самые обычные объекты. Наиболее уместно рассматривать массивы как «подтип» объектов (см. главу 3), в данном случае с дополнительными характеристиками числового индексирования (вместо обычных строковых ключей, как у простых объектов) и поддержания автоматически обновляемого свойства `.length`.

Значения как типы

В языке JavaScript у переменных нет типов — типы есть у *значений*. Переменная может хранить любое значение в любой момент времени.

Также на типы JS можно взглянуть под другим углом: в JS нет «принудительного контроля типов». Иначе говоря, движок не требует, чтобы в *переменной* всегда хранились переменные *исходного типа*, с которым она начала свое существование. В одной команде присваивания переменной может быть присвоена `строка`, в другой — `число`, и т. д.

Значение `42` обладает внутренним типом `number`, и этот *тип* изменить не удастся. Другое значение — например, `"42"` с типом `string` — может быть создано *на основе* значения `42` типа `number`, для чего используется процесс, называемый *преобразованием типов* (см. главу 4).

Применение `typeof` к переменной не означает «к какому типу относится эта переменная», как могло бы показаться на первый взгляд, потому что переменные JS не обладают типами. Вместо этого вы спрашиваете: «К какому типу относится значение в этой переменной?»

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

Оператор `typeof` всегда возвращает строку. Таким образом:

```
typeof typeof 42; // "string"
```

Первый оператор `typeof 42` возвращает `"number"`, а `typeof "number"` возвращает `"string"`.

undefined и необъявленные переменные

Переменные, *на данный момент* не имеющие значения, на самом деле имеют значение `undefined`. Вызов `typeof` для таких переменных возвращает `"undefined"`:

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// В будущем
b = c;
typeof b; // "undefined"
typeof c; // "undefined"
```

Многим разработчикам удобно рассматривать слово `"undefined"` как синоним для необъявленной переменной. Тем не менее в JS эти две концепции заметно отличаются.

Неопределенная переменная (`undefined`) была объявлена в доступной области видимости, но на данный момент не содержит никакого другого значения. С другой стороны, необъявленная переменная не была формально объявлена в доступной области видимости.

Пример:

```
var a;

a; // undefined
b; // ReferenceError: значение b не определено
```

Путаница возникает из-за сообщения об ошибке, которое выдается браузерами по этому условию. Как видите, выдается сообщение «значение `b` не определено» (`b is not defined`), очень легко и даже логично спутать его с «переменная `b` содержит `undefined`». Тем не менее `undefined` и «не определено» (not defined) — это совершенно разные понятия. Конечно, лучше бы браузеры выдавали сообщение вида «значение `b` не найдено» или «значение `b` не объявлено» для предотвращения путаницы!

Также `typeof` проявляет для необъявленных переменных специальное поведение, которое только усиливает путаницу.

Пример:

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

Оператор `typeof` возвращает `"undefined"` даже для «необъявленных» переменных. Обратите внимание: при выполнении `typeof b` не было выдано сообщения об ошибке, хотя переменная `b` и не объявлена. Это специальная защита для `typeof`.

Как я уже говорил, было бы лучше, если бы `typeof` для необъявленных переменных возвращал специальную строку `"undeclared"`, вместо того чтобы объединять возвращаемое значение с совершенно другим случаем `"undefined"`.

typeof для необъявленных переменных

Впрочем, эта защита может быть полезной при использовании JavaScript в браузерах, где несколько разных файлов сценариев могут загружать переменные в общее глобальное пространство имен.

Многие разработчики полагают, что в глобальном пространстве имен не должно быть никаких переменных, а все переменные должны содержаться в модулях и приватных/раздельных пространствах имен. Теоретически идея выглядит прекрасно, но на практике это невозможно; однако это достойная цель, к которой следует стремиться! К счастью, в ES6 добавилась полноценная поддержка модулей, благодаря которой такое разделение со временем станет куда более реальным.

Простой пример: представьте, что в вашей программе предусмотрен «режим отладки», который включается глобальной переменной (флагом) с именем `DEBUG`. Прежде чем выводить в консоль сообщение, нужно проверить, была ли эта переменная объявлена. Высокоуровневое объявление глобальной переменной `var DEBUG = true` будет включено только в файл debug.js, который загружается в браузере только в ходе разработки/тестирования, но не в среде реальной эксплуатации.

Вы должны внимательно отнестись к проверке глобальной переменной `DEBUG` в остальном коде приложения, чтобы избежать ошибки `ReferenceError`. В этом случае защита `typeof` приходит на помощь:

```
// Упс! Это вызовет ошибку!
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// Безопасная проверка существования
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

Подобные проверки полезны даже в том случае, если вы не работаете с переменными, определяемыми пользователем (как в случае с `DEBUG`). Если вы проводите проверку для встроенного API, возможно, также будет полезно сделать это без выдачи ошибки:

```
if (typeof atob === "undefined") {
    atob = function() { /*..*/ };
}
```

Если вы определяете «полифил» (polyfill) для функции, которая еще не существует, вероятно, лучше избежать использования `var` для объявления `atob`. Если вы объявите `var atob` внутри команды `if`, это объявление «поднимается» вверх области видимости, даже если условие `if` не проходит (потому что глобальная версия `atob` уже существует!). В некоторых браузерах и для некоторых специальных типов глобальных встроенных переменных (часто называемых «объектами-хостами») этот дубликат объявления может вызвать ошибку. Отсутствие `var` предотвращает поднятие объявления.

Также существует другой способ выполнения тех же проверок глобальных переменных без защиты `typeof`: проверка того, что все глобальные переменные также являются свойствами глобального объекта, которым в браузере фактически является объект `window`. Таким образом, те же проверки можно (вполне безопасно) выполнить следующим образом:

```
if (window.DEBUG) {
    // ..
}

if (!window.atob) {
    // ..
}
```

В отличие от обращений к необъявленным переменным, при попытке обратиться к несуществующему свойству объекта (даже глобального) ошибка `ReferenceError` не выдается.

С другой стороны, ручное обращение к глобальной переменной по имени `window` — практика, которой некоторые разработчики предпочитают избегать, особенно если код должен выполняться в разных средах JS (не только браузерах, но и в node.js на стороне

сервера, например), в которых глобальная переменная не всегда называется `window`.

С технической точки зрения защита `typeof` полезна, даже если вы не используете глобальные переменные, хотя эти обстоятельства встречаются реже, а некоторые разработчики считают это решение менее желательным. Представьте, что вы написали вспомогательную функцию, которая должна копироваться другими программистами в их программы и модули, и теперь вы хотите узнать, определила ли вызывающая программа некоторую переменную (и тогда ее можно использовать) или нет.

```
function doSomethingCool() {
    var helper =
        (typeof FeatureXYZ !== "undefined") ?
        FeatureXYZ :
        function() { /*.. действия по умолчанию ..*/ };

    var val = helper();
    // ..
}
```

`DoSomethingCool()` проверяет переменную с именем `FeatureXYZ`. Если переменная будет найдена, то она используется, а если нет — использует собственную версию. Если кто-то включит эту функцию в свой модуль/программу, то функция безопасно проверит, была определена переменная `FeatureXYZ` или нет:

```
// IIFE (см. "Немедленно вызываемые функциональные выражения"
// в книге "Область видимости и замыкания" этой серии)
(function(){
    function FeatureXYZ() { /*.. моя функциональность XYZ ..*/ }

    // include `doSomethingCool(..)`
    function doSomethingCool() {
        var helper =
            (typeof FeatureXYZ !== "undefined") ?
            FeatureXYZ :
            function() { /*.. действия по умолчанию ..*/ };

        var val = helper();
```

```
        // ..
    }

    doSomethingCool();
})();
```

Здесь `FeatureXYZ` не является глобальной переменной, но мы все равно используем защиту `typeof`, чтобы обезопасить ее проверку. И что важно, здесь *нет* объекта, который можно было бы использовать для проверки (как это делалось для глобальных переменных с `window.___`), так что `typeof` здесь сильно помогает.

Другие разработчики предпочитают паттерн проектирования, называемый «внедрением зависимостей». В этом случае вместо неявной проверки определения `FeatureXYZ` где-то снаружи, `doSomethingCool()` требует явной передачи зависимости:

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. Действия по умолчанию ..*/ };

    var val = helper();
    // ..
}
```

Существует много вариантов проектирования подобной функциональности. Ни один паттерн не может быть «правильным» или «ошибочным», у каждого есть свои плюсы и минусы. Однако в целом довольно удобно, что защита `typeof` для необъявленных переменных предоставляет нам больше возможностей.

Итоги

В JavaScript есть семь встроенных *типов*: `null`, `undefined`, `boolean`, `number`, `string`, `object` и `symbol`. Тип значений можно идентифицировать оператором `typeof`.

Переменные не имеют типов, но зато типы есть у хранящихся в них значений. Типы определяют поведение, присущее этим значениям.

Многие разработчики считают, что «не определено» (`undefined`) и «не объявлено» — приблизительно одно и то же, но в JavaScript эти понятия заметно отличаются. `Undefined` — значение, которое может храниться в объявленной переменной. «Не объявлено» означает, что переменная не была объявлена.

К сожалению, JavaScript отчасти объединяет эти два термина, не только в сообщениях об ошибках («ReferenceError: a is not defined»), но и в возвращаемых значениях оператора `typeof`, который в обоих случаях возвращает `"undefined"`.

Тем не менее защита `typeof` (предотвращение ошибки) при использовании с необъявленной переменной бывает достаточно полезной в некоторых случаях.

2 Значения

Массивы, строки и числа — основные структурные элементы любой программы. Однако в JavaScript эти типы обладают некоторыми уникальными особенностями, которые могут как порадовать вас, так и привести в замешательство.

Давайте рассмотрим некоторые встроенные типы значений в JS и разберемся, как лучше понять и использовать особенности их поведения.

Массивы

В отличие от других языков с принудительным контролем типов, массивы JavaScript представляют собой контейнеры для значений любого типа, от `string` до `number` и `object`, и даже другого `массива` (именно так создаются многомерные `массивы`):

```
var a = [ 1, "2", [3] ];

a.length;       // 3
a[0] === 1;     // true
a[2][0] === 3;  // true
```