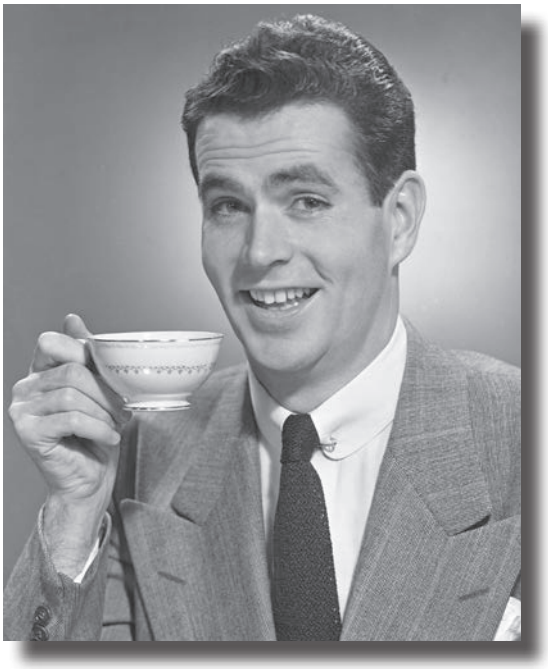
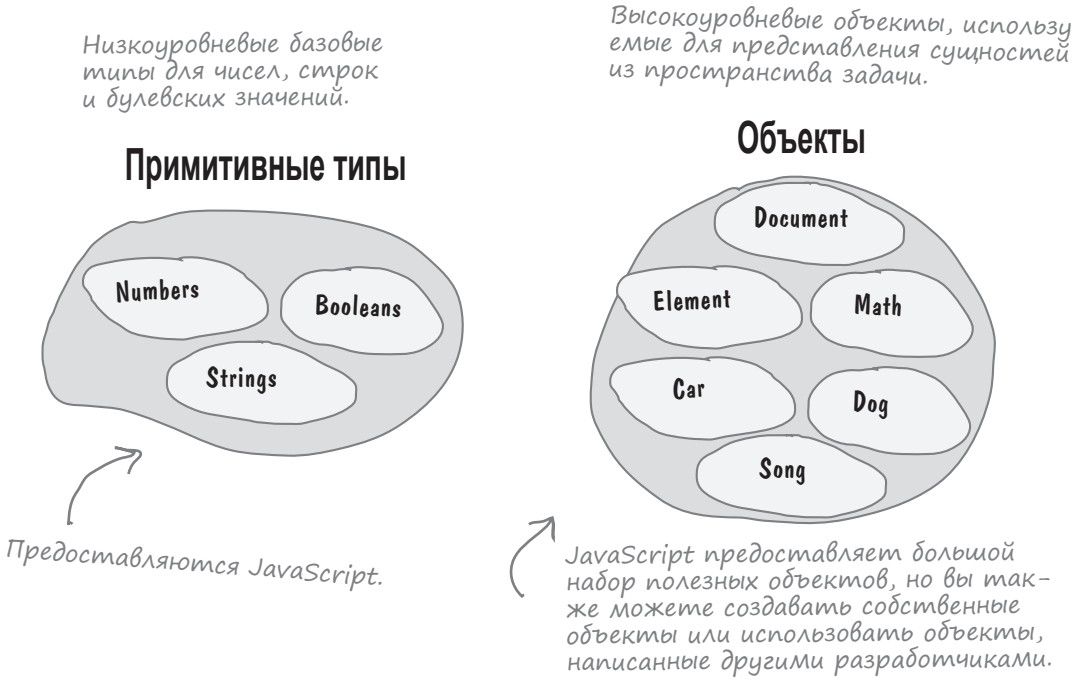
Серьезные типы



Настало время серьезно поговорить о типах. Одна из замечательных особенностей JavaScript заключается в том, что начинающий может достаточно далеко продвинуться, не углубляясь в подробности языка. Но чтобы действительно **овладеть языком**, получить повышение по работе и заняться тем, чем действительно стоит заниматься, нужно хорошо разбираться в **типах**. Помните, что мы говорили о JavaScript, — что у этого языка не было такой роскоши, как академическое определение, прошедшее экспертную оценку? Да, это правда, но отсутствие академической основы не помешало Стиву Джобсу и Биллу Гейтсу; не помешало оно и JavaScript. Это означает, что система типов JavaScript... ну, скажем так — не самая продуманная, и в ней найдется немало **странностей**. Но не беспокойтесь, в этой главе мы все разберем, и вскоре вы научитесь благополучно обходить все эти неприятные моменты с типами.

Истина где-то рядом...

У вас уже есть немалый опыт работы с типами JavaScript — это и примитивы с числами, строками и булевскими значениями, и многочисленные объекты, одни из которых предоставляет JavaScript (как, например, объект Math), другие предоставляет браузер (как объект document), а третьи вы написали самостоятельно. Казалось бы, дальше можно просто нежиться под теплым сиянием простой, мощной и последовательной системы типов JavaScript.



В конце концов, чего вы ожидали от официального языка веб-программирования? Будь вы простым сценарным программистом, вы могли бы откинуться в кресле, потягивая мартини, и наслаждаться желанным отдыхом...

Но вы не простой сценарный программист, и вы чувствуете, что здесь что-то не так. У вас возникает тревожное чувство, что за заборами Объективиля происходит что-то странное. До вас доносились слухи о строках, которые ведут себя как объекты; вы читали в блогах о типа null (возможно, небезопасном для здоровья); люди шепотом говорят, что интерпретатор JavaScript в последнее время выполняет какие-то странные преобразования типов. Что все это значит? Мы не знаем, но истина где-то рядом, и мы откроем ее в этой главе. А когда это произойдет, ваши представления о том, что есть истина, а что есть ложь, могут резко измениться.

Группа значений JavaScript и незваных гостей, облачившись в маскарадные костюмы, развлекаются игрой «Кто я?». Они дают подсказки, а вы должны угадать их по тому, что они говорят о себе. Предполагается, что участники всегда говорят о себе правду. Соедините линией каждое высказывание с именем соответствующего участника. Мы уже провели одну линию за вас. Прежде чем двигаться дальше, сверьтесь с ответами в конце главы.

Если упражнение покажется вам слишком сложным — не отчаивайтесь и подсмотрите ответ.

Я возвращаюсь из функции при отсутствии команды return.

Я считаюсь значением переменной, которой еще не было присвоено значение.

Я — значение элемента, не существующего в разреженном массиве.

Я — значение несуществующего свойства.

Я — значение удаленного свойства.

Я — значение, которое не может быть задано свойству при создании объекта.

Сегодняшние участники:

нуль

пустой объект

null

undefined

NaN

infinity

area 51

...---...

{}

[]

Будьте осторожны: undefined иногда появляется совершенно неожиданно...

Каждый раз, когда что-то идет не так, вы используете переменную, которая еще не была инициализирована, пытаетесь получить значение несуществующего (или удаленного) свойства, обращаетесь к отсутствующему элементу массива — вы сталкиваетесь с `undefined`.

Что это такое, спросите вы? Ничего сложного. Считайте, что `undefined` — это значение, которое присваивается тому, что еще не имеет значения (другими словами, не было инициализировано).

Какая польза от undefined? Оно позволяет проверить, было ли присвоено значение переменной (или свойству, или элементу массива). Рассмотрим пару примеров начиная с неинициализированной переменной.

Мы можем проверить, была ли инициализирована переменная. Просто сравните ее с undefined.

```
var x;

if (x == undefined) {
    // Переменная x не инициализирована! Принять меры!
}
```

Обратите внимание: здесь используется значение undefined. Не путайте его со строкой "undefined".

Теперь проверяем свойство объекта:

Чтобы проверить свойство на неопределенность, также сравните его со значением undefined.

```
var customer = {
    name: "Jenny"
};
if (customer.phoneNumber == undefined) {
    // Получить телефон клиента
}
```

Часто Задаваемые Вопросы

В: Когда нужно проверять переменную (свойство, элемент массива) на неопределенность?

О: Это зависит от структуры кода. Если ваш код написан так, что свойство или переменная может не иметь значения при выполнении некоторого блока, то проверка на undefined даст вам возможность обработать эту ситуацию вместо того, чтобы выполнять вычисления с неопределенными значениями.

В: Если undefined — это значение, то есть ли у него тип?

О: Да, есть — значение undefined относится к типу Undefined. Почему? Вероятно, логика выглядит примерно так: это не объект, не число, не строка и не булевское значение... И вообще ничто определенное. Так почему бы не назначить этой неопределенности «неопределенный» тип? Это одна из тех странностей JavaScript, с которыми нужно просто смириться.

В ЛАБОРАТОРИИ

В своей лаборатории мы любим разбирать, заглядывать «под капот», экспериментировать, подключать диагностические инструменты и выяснять, что же происходит на самом деле. Сегодня, в ходе исследования системы типов JavaScript, мы обнаружили маленький диагностический инструмент для анализа переменных — **typeof**. Наденьте лабораторный халат и защитные очки, заходите и присоединяйтесь к нам.

Оператор **typeof** встроен в JavaScript. Он используется для проверки типа операнда (того, к чему применяется оператор). Пример:

```
var subject = "Just a string";

var probe = typeof subject;
console.log(probe);
```

Оператор typeof получае операнд и определяет его тип.

В данном случае выводится тип string. Обратите внимание: для представления типов typeof использует строки вида "string", "boolean", "number", "object", "undefined" и т. д.

Консоль JavaScript

```
string
```

Теперь ваша очередь. Соберите данные по следующим экспериментам:

```
var test1 = "abcdef";
var test2 = 123;
var test3 = true;
var test4 = {};
var test5 = [];
var test6;
var test7 = {"abcdef": 123};
var test8 = ["abcdef", 123];
function test9(){return "abcdef"};

console.log(typeof test1);
console.log(typeof test2);
console.log(typeof test3);
console.log(typeof test4);
console.log(typeof test5);
console.log(typeof test6);
console.log(typeof test7);
console.log(typeof test8);
console.log(typeof test9);
```

Тестовые данные и тесты.

Консоль JavaScript

Запишите здесь результаты. Какие-нибудь сюрпризы?

Да, здесь часто возникает путаница. Во многих языках существует концепция значения, представляющего «отсутствие объекта». И это вполне нормально — возьмем хотя бы метод `document.getElementById`. Он ведь должен возвращать объект, верно? А что произойдет, если он не сможет вернуть объект? Тогда он должен вернуть какой-то признак, означающий: «Здесь мог бы быть объект, но, к сожалению, его нет». Именно этот смысл заложен в `null`.

Переменной также можно явно присвоить `null`:

```
var killerObjectSomeday = null;
```

Что может означать присваивание `null` переменной? Например, «Когда-нибудь в будущем мы присвоим объект этой переменной, но пока еще не присвоили».

Если в этот момент вы недоумеваете и спрашиваете: «Хмм, а почему они не использовали `undefined`?» — знайте, что вы не одни. Это решение было принято в самом начале существования JavaScript. Разработчики хотели иметь одно значение для переменных, которые еще не были инициализированы, и другое для обозначения отсутствия объекта. Может, решение не самое красивое и немного избыточное, но дело обстоит именно так. Просто запомните смысл каждого из значений (`undefined` и `null`) и знайте, что null чаще всего используется там, где объект еще не создан или не найден, а `undefined` — для неинициализированных переменных, отсутствующих свойств объектов или отсутствующих значений в массивах.

Снова в лабораторию

Стоп, мы забыли включить null в свои тестовые данные. Недостающий тест:

```
var test10 = null;

console.log(typeof test10);
```

Запишите здесь свой результат.

Консоль JavaScript

Как использовать null

Существует великое множество функций и методов, возвращающих объекты. Часто бывает нужно убедиться в том, что вы получаете полноценный объект, а не `null`, — на случай, если функция не смогла найти или создать возвращаемый объект. Примеры уже встречались нам при работе с DOM:

```
var header = document.getElementById("header");
if (header == null) {
     // Заголовка нет — какая-то серьезная проблема!
}
```

Ищем элемент header, без которого никак не обойтись.

Ну и ну, заголовок не существует! Спасайся, кто может!

Следует учитывать, что получение `null` не всегда означает, что что-то пошло не так. Это может означать, что объект еще не существует и его нужно создать или его нужно пропустить при обработке. Пользователь может открыть или закрыть погодный виджет на сайте. Если виджет открыт, то в DOM существует элемент `<div>` с идентификатором "weatherDiv", а если закрыт — такого элемента нет. Так польза от `null` становится очевидной:

Проверим, существует ли элемент с идентификатором "weatherDiv".

```
var weather = document.getElementById("weatherDiv");
if (weather != null) {
     // Создание содержимого для погодного виджета
}
```

Если результат getElementById отличен от null, значит, на странице существует такой элемент. Создадим для него погодный виджет (который, как предполагается, загружает местную сводку погоды).

При помощи null можно проверить, существует объект или нет.

Помните: значение null используется для представления несуществующих объектов.

Хотите верьте, хотите нет!

Число которое не является числом

Легко написать команду JavaScript, результатом которой является неопределенное числовое значение.

Несколько примеров:

```
var a = 0/0;
```

В математике это выражение не имеет однозначного результата — так откуда его возьмет JavaScript?

```
var b = "food" * 1000;
```

Мы не знаем, как должен выглядеть результат, но это безусловно не число!

```
var c = Math.sqrt(-9);
```

Квадратный корень из отрицательного числа — это комплексное число, а в JavaScript такие числа не имеют представления.

Хотите верьте, хотите нет, но существуют числовые значения, которые **невозможно представить** в JavaScript! В JavaScript эти значения не выражаются, поэтому для них используется специальное значение:

NaN

В JavaScript используется значение NaN (сокращение от "Not a Number", то есть «не число») для представления числовых результатов... не имеющих представления. Для примера возьмем 0/0. Результат 0/0 не имеет собственного представления на компьютере, поэтому в JavaScript он представляется специальным значением NaN.

ВОЗМОЖНО, NaN — САМОЕ СТРАННОЕ ЗНАЧЕНИЕ В МИРЕ. Оно не только представляет все числовые значения, не имеющие собственного представления; это единственное значение в JavaScript, не равное самому себе!

Да, вы поняли правильно. Если сравнить NaN с NaN, они не будут равны!

NaN != NaN

Работа с NaN

Может показаться, что вам очень редко придется иметь дело со значением `NaN`, но если ваш код обрабатывает числовые данные, вы удивитесь, как часто оно будет встречаться на вашем пути. Вероятно, самой частой операцией будет проверка числа на NaN. С учетом того, что вы узнали о JavaScript, решение может показаться очевидным:

ОШИБКА!

```
if (myNum == NaN) {
    myNum = 0;
}
```

Вроде бы должно работать... Но не работает.

Логично предположить, что проверять переменную на хранение значения NaN нужно именно так... Но это решение не работает. Почему? Потому что значение NaN не равно ничему, даже самому себе, значит, любые проверки равенства с NaN исключаются. Вместо этого приходится использовать специальную функцию `isNaN`. Это делается так:

ПРАВИЛЬНО!

```
if (isNaN(myNum)) {
    myNum = 0;
}
```

Функция isNaN возвращает true, если переданное ей значение не является числом.

А дальше еще удивительнее

Итак, давайте немного подумаем. Если `NaN` означает «не число», то что это? Разве не проще было бы выбрать имя по тому, чем оно является (а не по тому, чем оно не является)? А как вы думаете, чем оно является? Тип NaN можно проверить оператором typeof с совершенно неожиданным удивительным результатом:

```
var test11 = 0 / 0;
console.log(typeof test11);
```

Вот что мы получаем.

Если у вас голова не идет кругом, вероятно, вы чего-то не поняли из этой книги.

Что вообще творится? `NaN` имеет числовой тип? Как то, что не является числом, может иметь числовой тип? Спокойно, дышите глубже. Считайте, что имя `NaN` просто выбрано неудачно. Вероятно, вместо «не число» стоило использовать что-то вроде «число, не имеющее представления» (хотя, конечно, красивого сокращения уже не получится). Лучше всего рассматривать `NaN` именно так — как число, которое невозможно представить (по крайней мере на компьютере).

В общем, ваш список странностей JavaScript расширяется.