O'REILLY®

"An excellent look at the core JavaScript fundamentals that copy and paste and JavaScript toolkits don't and could never teach you."
–**DAVID WALSH**, Senior Web Developer, Mozilla

KYLE SIMPSON

TYPES & GRAMMAR

Free Sampler

YOU DON'T KNOW JS

THE *YOU DON'T KNOW JS* SERIES INCLUDES:

Up & Going
Scope & Closures
this & Object Prototypes
Types & Grammar
Async & Performance
ES6 & Beyond

TYPES & GRAMMAR

No matter how much experience you have with JavaScript, odds are you don't fully understand the language. As part of the *You Don't Know JS* series, this compact guide explores JavaScript types in greater depth than previous treatments by looking at type coercion problems, demonstrating why types work, and showing you how to take advantage of these features.

Like other books in this series, *You Don't Know JS: Types & Grammar* dives into trickier parts of the language that many JavaScript programmers simply avoid or assume don't exist (like types). Armed with this knowledge, you can achieve true JavaScript mastery.

WITH THIS BOOK YOU WILL:

- **Get acquainted with JavaScript's seven types:** `null`, `undefined`, `boolean`, `number`, `string`, `object`, **and** `symbol`
- **Understand why JavaScript's unique array, string, and number characteristics may delight or confound you**
- **Learn how natives provide object wrappers around primitive values**
- **Dive into the coercion controversy—and learn why this feature is useful in many cases**
- **Explore various nuances in JavaScript syntax, involving statements, expressions, and other features**

KYLE SIMPSON is an Open Web evangelist who's passionate about all things JavaScript. He's an author, workshop trainer, tech speaker, and OSS contributor/leader.

JAVASCRIPT

US $24.99 **CAN $28.99**

ISBN: 978-1-491-90419-0

9 781491 904190 52499

O'REILLY®

oreilly.com

YouDontKnowJS.com

You Don't Know JS: Types & Grammar
by Kyle Simpson

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Simon St. Laurent and Brian MacDonald
Production Editor: Kristen Brown
Copyeditor: Christina Edwards
Proofreader: Charles Roumeliotis
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen

February 2015: First Edition

Revision History for the First Edition
2015-01-23: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491904190* for release details.

978-1-491-90419-0

[LSI]

Table of Contents

CHAPTER 1
Types

Most developers would say that a dynamic language (like JS) does not have *types*. Let's see what the ES5.1 specification (*http://www.ecma-international.org/ecma-262/5.1/*) has to say on the topic:

> Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this clause. Types are further sub-classified into ECMAScript language types and specification types.
>
> An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Number, and Object.

Now, if you're a fan of strongly typed (statically typed) languages, you may object to this usage of the word "type." In those languages, "type" means a whole lot *more* than it does here in JS.

Some people say JS shouldn't claim to have "types," and they should instead be called "tags" or perhaps "subtypes."

Bah! We're going to use this rough definition (the same one that seems to drive the wording of the spec): a *type* is an intrinsic, built-in set of characteristics that uniquely identifies the behavior of a particular value and distinguishes it from other values, both to the engine *and* to the developer.

In other words, if both the engine and the developer treat value `42` (the number) differently than they treat value `"42"` (the string), then those two values have different *types*—`number` and `string`, respec-

tively. When you use `42`, you are *intending* to do something numeric, like math. But when you use `"42"`, you are *intending* to do something string'ish, like outputting to the page, etc. These two values have different types.

That's by no means a perfect definition. But it's good enough for this discussion. And it's consistent with how JS describes itself.

A Type by Any Other Name…

Beyond academic definition disagreements, why does it matter if JavaScript has *types* or not?

Having a proper understanding of each *type* and its intrinsic behavior is absolutely essential to understanding how to properly and accurately convert values to different types (see Chapter 4). Nearly every JS program ever written will need to handle value coercion in some shape or form, so it's important you do so responsibly and with confidence.

If you have the `number` value `42`, but you want to treat it like a `string`, such as pulling out the `"2"` as a character in position `1`, you obviously must first convert (coerce) the value from `number` to `string`.

That seems simple enough.

But there are many different ways that such coercion can happen. Some of these ways are explicit, easy to reason about, and reliable. But if you're not careful, coercion can happen in very strange and surprising ways.

Coercion confusion is perhaps one of the most profound frustrations for JavaScript developers. It has often been criticized as being so *dangerous* as to be considered a flaw in the design of the language, to be shunned and avoided.

Armed with a full understanding of JavaScript types, we're aiming to illustrate why coercion's *bad reputation* is largely overhyped and somewhat undeserved—to flip your perspective so you see coercion's power and usefulness. But first, we have to get a much better grip on values and types.

Built-in Types

JavaScript defines seven built-in types:

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol`—added in ES6!

All of these types except `object` are called "primitives."

The `typeof` operator inspects the type of the given value, and always returns one of seven string values—surprisingly, there's not an exact 1-to-1 match with the seven built-in types we just listed:

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

// added in ES6!
typeof Symbol()      === "symbol";    // true
```

These six listed types have values of the corresponding type and return a string value of the same name, as shown. `Symbol` is a new data type as of ES6, and will be covered in Chapter 3.

As you may have noticed, I excluded `null` from the above listing. It's *special*—special in the sense that it's buggy when combined with the `typeof` operator:

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

It would have been nice (and correct!) if it returned `"null"`, but this original bug in JS has persisted for nearly two decades, and will

likely never be fixed because there's so much existing web content that relies on its buggy behavior that "fixing" the bug would *create* more "bugs" and break a lot of web software.

If you want to test for a `null` value using its type, you need a compound condition:

```
var a = null;

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

`null` is the only primitive value that is "falsy" (aka false-like; see Chapter 4) but which also returns `"object"` from the `typeof` check.

So what's the seventh string value that `typeof` can return?

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

It's easy to think that `function` would be a top-level built-in type in JS, especially given this behavior of the `typeof` operator. However, if you read the spec, you'll see it's actually somewhat of a "subtype" of `object`. Specifically, a function is referred to as a "callable object"—an object that has an internal `[[Call]]` property that allows it to be invoked.

The fact that functions are actually objects is quite useful. Most importantly, they can have properties. For example:

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

The function object has a `length` property set to the number of formal parameters it is declared with:

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

Since you declared the function with two formal named parameters (`b` and `c`), the "length of the function" is `2`.

What about arrays? They're native to JS, so are they a special type?

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

Nope, just objects. It's most appropriate to think of them also as a "subtype" of `object` (see Chapter 3), in this case with the additional characteristics of being numerically indexed (as opposed to just being string-keyed like plain objects) and maintaining an automatically updated `.length` property.

Values as Types

In JavaScript, variables don't have types—*values have types*. Variables can hold any value, at any time.

Another way to think about JS types is that JS doesn't have "type enforcement," in that the engine doesn't insist that a *variable* always holds values of the *same initial type* that it starts out with. A variable can, in one assignment statement, hold a `string`, and in the next hold a `number`, and so on.

The *value* `42` has an intrinsic type of `number`, and its *type* cannot be changed. Another value, like `"42"` with the `string` type, can be created *from* the `number` value `42` through a process called *coercion* (see Chapter 4).

If you use `typeof` against a variable, it's not asking "What's the type of the variable?" as it may seem, since JS variables have no types. Instead, it's asking "What's the type of the value *in* the variable?"

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

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

The `typeof` operator always returns a string. So:

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

The first `typeof 42` returns `"number"`, and `typeof "number"` is `"string"`.

undefined Versus "undeclared"

Variables that have no value *currently* actually have the `undefined` value. Calling `typeof` against such variables will return `"undefined"`:

```
var a;

typeof a; // "undefined"

var b = 42;
var c;

// later
b = c;
```

```
typeof b; // "undefined"
typeof c; // "undefined"
```

It's tempting for most developers to think of the word "undefined" as a synonym for "undeclared." However, in JS, these two concepts are quite different.

An "undefined" variable is one that has been declared in the accessible scope, but *at the moment* has no other value in it. By contrast, an "undeclared" variable is one that has not been formally declared in the accessible scope.

Consider:

```
var a;

a; // undefined
b; // ReferenceError: b is not defined
```

An annoying confusion is the error message that browsers assign to this condition. As you can see, the message is "b is not defined," which is of course very easy and reasonable to confuse with "b is undefined." Yet again, "undefined" and "is not defined" are very different things. It'd be nice if the browsers said something like "b is not found" or "b is not declared" to reduce the confusion!

There's also a special behavior associated with `typeof` as it relates to undeclared variables that even further reinforces the confusion. Consider:

```
var a;

typeof a; // "undefined"

typeof b; // "undefined"
```

The `typeof` operator returns `"undefined"` even for "undeclared" (or "not defined") variables. Notice that there was no error thrown when we executed `typeof b`, even though `b` is an undeclared variable. This is a special safety guard in the behavior of `typeof`.

Similar to above, it would have been nice if `typeof` used with an undeclared variable returned "undeclared" instead of conflating the result value with the different "undefined" case.

typeof Undeclared

Nevertheless, this safety guard is a useful feature when dealing with JavaScript in the browser, where multiple script files can load variables into the shared global namespace.

Many developers believe there should never be any variables in the global namespace, and that everything should be contained in modules and private/separate namespaces. This is great in theory but nearly impossible in practice; still, it's a good goal to strive toward! Fortunately, ES6 added first-class support for modules, which will eventually make that much more practical.

As a simple example, imagine having a "debug mode" in your program that is controlled by a global variable (flag) called `DEBUG`. You'd want to check if that variable was declared before performing a debug task like logging a message to the console. A top-level global `var DEBUG = true` declaration would only be included in a "debug.js" file, which you only load into the browser when you're in development/testing, but not in production.

However, you have to take care in how you check for the global `DEBUG` variable in the rest of your application code, so that you don't throw a `ReferenceError`. The safety guard on `typeof` is our friend in this case:

```
// oops, this would throw an error!
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// this is a safe existence check
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

This sort of check is useful even if you're not dealing with user-defined variables (like `DEBUG`). If you are doing a feature check for a built-in API, you may also find it helpful to check without throwing an error:

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

If you're defining a "polyfill" for a feature if it doesn't already exist, you probably want to avoid using `var` to make the `atob` declaration. If you declare `var atob` inside the `if` statement, this declaration is hoisted (see the *Scope & Closures* title in this series) to the top of the scope, even if the `if` condition doesn't pass (because the global `atob` already exists!). In some browsers and for some special types of global built-in variables (often called "host objects"), this duplicate declaration may throw an error. Omitting the `var` prevents this hoisted declaration.

Another way of doing these checks against global variables but without the safety guard feature of `typeof` is to observe that all global variables are also properties of the global object, which in the browser is basically the `window` object. So, the above checks could have been done (quite safely) as:

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

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

Unlike referencing undeclared variables, there is no `ReferenceError` thrown if you try to access an object property (even on the global `window` object) that doesn't exist.

On the other hand, manually referencing the global variable with a `window` reference is something some developers prefer to avoid, especially if your code needs to run in multiple JS environments (not just browsers, but server-side node.js, for instance), where the global variable may not always be called `window`.

Technically, this safety guard on `typeof` is useful even if you're not using global variables, though these circumstances are less common, and some developers may find this design approach less desirable. Imagine a utility function that you want others to copy-and-paste into their programs or modules, in which you want to check to see if the including program has defined a certain variable (so that you can use it) or not:

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

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

`doSomethingCool()` tests for a variable called `FeatureXYZ`, and if found, uses it, but if not, uses its own. Now, if someone includes this utility into their module/program, it safely checks if they've defined `FeatureXYZ` or not:

```
// an IIFE (see the "Immediately Invoked Function Expressions"
// discussion in the Scope & Closures title in this series)
(function(){
    function FeatureXYZ() { /*.. my XYZ feature ..*/ }

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

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

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

Here, `FeatureXYZ` is not at all a global variable, but we're still using the safety guard of `typeof` to make it safe to check for. And importantly, here there is *no* object we can use (like we did for global variables with `window.___`) to make the check, so `typeof` is quite helpful.

Other developers would prefer a design pattern called "dependency injection," where instead of `doSomethingCool()` inspecting implicitly for `FeatureXYZ` to be defined outside/around it, it would need to have the dependency explicitly passed in, like:

```
function doSomethingCool(FeatureXYZ) {
    var helper = FeatureXYZ ||
        function() { /*.. default feature ..*/ };

    var val = helper();
```

```
    // ..
}
```

There's lots of options when designing such functionality. No one pattern here is "correct" or "wrong"—there are various trade-offs to each approach. But overall, it's nice that the `typeof` undeclared safety guard gives us more options.

Review

JavaScript has seven built-in *types*: `null`, `undefined`, `boolean`, `number`, `string`, `object`, and `symbol`. They can be identified by the `typeof` operator.

Variables don't have types, but the values in them do. These types define the intrinsic behavior of the values.

Many developers will assume "undefined" and "undeclared" are roughly the same thing, but in JavaScript, they're quite different. `undefined` is a value that a declared variable can hold. "Undeclared" means a variable has never been declared.

JavaScript unfortunately kind of conflates these two terms, not only in its error messages ("ReferenceError: a is not defined") but also in the return values of `typeof`, which is `"undefined"` for both cases.

However, the safety guard (preventing an error) on `typeof` when used against an undeclared variable can be helpful in certain cases.