Riverside JS Workshop
JavaScript Skills Ladder, Lesson 3.2

JavaScript Data Types

Lesson Objectives

When you complete this lesson, you will be able to:

- *differentiate primitives and objects.*
- *categorize primitive types.*
- *distinguish null and undefined.*
- *construct objects.*
- *use the console to experiment with JavaScript types, object properties, and various number representations.*

JavaScript only has a few basic types, but they still require consideration. In this lesson, we'll review the basics of primitives and objects, delve into a few details you may not have encountered before, and deepen your understanding of the fundamentals.

Know your Types: Primitives and Objects

Every value in JavaScript is either a primitive or an object. Primitives are simple types, like numbers and strings, while objects are complex types because they are composed of multiple values, like this `square` object:

OBSERVE:

```
var square = {
    width: 10,
    height: 10
};
```

This object is composed of two primitive values: a width, with a value of 10, and a height, that also has the value of 10. We'll look into primitives first. We'll come back to objects later.

Primitives

The three primitive types you'll work with most often are *numbers*, *strings*, and *booleans*. You can test these types right in the browser's console. Open the console in a browser window and try typing in some primitive values. Some browsers don't allow you to open the console for an empty page. You can load the web page you created for *basics.html* in the previous lesson, and then open the console, and trying testing some types, like this:

INTERACTIVE SESSION:

```
> 3
3
> "test"
"test"
> true
true
```

Let's try an expression:

INTERACTIVE SESSION:

```
> 3 + 5
8
```

When you type an expression, the result of that expression is a value, which is displayed in the console. When you type a statement, the result of that statement is (usually) `undefined`:

INTERACTIVE SESSION:

```
> var x = 3;
undefined
> x + 5
8
```

Here, we declared a new variable, `x`, in a statement (the first thing you typed into the console), and then used it in the expression `x + 5` (the second thing you typed into the console). Even though the statement sets the value of `x` to 3, the result of the statement itself is `undefined`. The result of the expression is just the value that the expression computes. Get used to this behavior so you don't get confused when you see `undefined` in the console!

Getting the Type of a Value with Typeof

JavaScript has a *typeof* operator that you can use to check the type of a value. There are a couple of reasons that you don't necessarily want to rely on this operator in your code. We'll get to those reasons a bit, but for now, we'll use *typeof* to check the type of our primitives, like this:

INTERACTIVE SESSION:

```
> typeof 3
"number"
> typeof x
"number"
> typeof "test"
```

```
"string"
> typeof true
"boolean"
```

The typeof operator might look a little strange at first. It's not like other operators you're used to that take two values. typeof takes just one value, and it returns the type of that value as a string. So the type of the number 3 is returned as the string "number." Notice that we can use typeof on either values (like 3) or variables containing values (like `x`). You can also use typeof in an expression, like this:

INTERACTIVE SESSION:

```
> if (typeof x == "number") {
      alert("x is a number!");
  }
undefined
```

Remember to use *Ctrl+Enter* or *Shift+Enter* at the end of a line in the console to avoid getting an error. Use *Enter* at the end of the `if` statement (after the closing curly brace, `}`). Do you get the alert?

We compare the result of the expression `typeof x` with the string `"number"`, and if they're equal, alerting a message (yes, you can alert from the console!). The result of the `if` statement is `undefined`.

Null and Undefined

Now, let's take a look at these primitive types: *null* and *undefined*. You've seen undefined as the result of statements you typed in the console. It pops up in other places as well, but let's begin with null. null is a way to say that a variable has "no value":

INTERACTIVE SESSION:

```
> var y = null;
undefined
> if (y == null) {
      console.log("y is null!");
  }
y is null!
< undefined
> typeof y
"object"
```

Here, we assigned the value `null` to the variable `y`. So `y` has a value—a value that means "no value." Weird. We can compare that value to `null`, and since `y` has the value `null`, that

comparision is true, and so we see the message `"y is null!"` in the console. (Yes, we can call `console.log()` from the console!) Notice that you see `"y is null"` in the console, and then you see the result of the statement, which is `undefined`. In Chrome and Safari, the console displays a little < character next to the undefined result of the statement so you don't mix up the output to the console (`"y is null!"`) with the result of the statement (`undefined`). In Firefox, you'll see a right-pointing arrow next to the undefined result of the statement.

Weirder still, when you check the `typeof y`, you get `"object"` as the result. What? That doesn't seem right. Well, guess what—it's *not*! This is an error in the current implementation of JavaScript. The result *should* be `null`, because the type of `null` is `null`.

Note This error is one reason you don't want to rely on *typeof* in your code. This mistake should be fixed in future implementations of JavaScript, but for now, just keep this in mind if you ever do need to use typeof. (There's one other issue with typeof we'll get to later).

Okay, so what about *undefined*? Lots of people confuse *null* with *undefined* when they first start learning JavaScript, so don't worry if it seems a bit murky. While null is a value that means "no value" (a mind bender in itself), *undefined* means that a variable has no value at all, not even null:

INTERACTIVE SESSION:

```
> var z;
undefined
> z
undefined
```

You can create an undefined variable by declaring it and not initializing it. Here, we declared the variable `z`, but didn't initialize it to a value. When we check the value of `z` by typing its name in the console, we get the result `undefined`.

Don't confuse the `undefined` you see as the result of the statement `var z;` with the `undefined` you see that is the result of the expression, `z`.

So what is the type of `undefined`? Can you guess? (This time, JavaScript has the *correct* implementation.)

INTERACTIVE SESSION:

```
> typeof z
undefined
```

Yes, the type of *undefined* is `undefined`!

Even though `z` is `undefined` (meaning that it has no value), you can test to see if `z` is undefined, like this:

INTERACTIVE SESSION:

```
> if (z == undefined) {
      console.log("z is undefined!");
  }
z is undefined!
< undefined
```

Or you can test it like this:

INTERACTIVE SESSION:

```
> if (typeof z == "undefined") {
      console.log("z is undefined!");
  }
z is undefined!
< undefined
```

You'll get the same result. Try it! Still, in general, we recommend that you don't use typeof unless you have a good reason. You can test a variable to see it if is undefined directly by comparing the value of the variable (in this case `z`, to the *value* `undefined`). You don't really need to use typeof at all here.

Some Interesting Numbers

Before we leave the primitive types, let's talk a little more about numbers. In your JavaScript programs, you've probably used numbers to loop over arrays, represent prices, count things, and much more, but there are a few numbers you might not have run into yet.

First, let's go over how numbers are represented. In JavaScript there are two ways to represent numbers: as integers and as floating point numbers. For example:

INTERACTIVE SESSION:

```
> var myInt = 3;
undefined
> var myFloat = 3.12583E03;
undefined
> myInt
3
> myFloat
3125.83
```

You can write a floating point number using scientific notation, `3.12583E03` (which means that the number after the "E" is the number of times you multiply the number by 10). This is handy when you want to represent very large or very small numbers.

Speaking of which, how do you know the largest or smallest numbers that you can represent? The JavaScript *Number* object (which we'll talk more about later) has built-in properties for both of these: `Number.MAX_VALUE` and `Number.MIN_VALUE`. Try them in your console:

INTERACTIVE SESSION:

```
> Number.MAX_VALUE
1.7976931348623157e+308
> Number.MIN_VALUE
5e-324
```

Wow. The `MAX_VALUE` is pretty large, and the `MIN_VALUE` is pretty small. You might think that means that JavaScript can represent a *lot* of numbers, but the actual number of numbers JavaScript can represent is much smaller than you might think. Why? Because both the `MAX_VALUE` and `MIN_VALUE` numbers are represented as floating point numbers and floating point numbers are not always precise. Notice that the largest value has only 17 decimals, which means it is only precise in the first 17 places. Beyond that number of places, it's all zeros, which means you couldn't accurately represent the number `1.797693134862315700000000 1e+308`, for instance. Floating point numbers are useful in some circumstances when you're working with big numbers *and* you don't need precision.

When you do need precision, you'll want to use integers, and you'll need to know the largest (and smallest) integer that JavaScript can represent. Let's take a look at the largest integer value in JavaScript, 2 raised to the power of 53:

INTERACTIVE SESSION:

```
> Math.pow(2, 53)
9007199254740992
```

This is a pretty big number too, but it's a lot smaller than `Number.MAX_VALUE`. JavaScript can represent all the integer values from zero up to this number *precisely*. That gives you a lot of numbers to play with, and it's unlikely that you'll ever need a larger number than this (this also applies to the smallest integer number, `Math.pow(2, -53)`).

Understanding how computers represent numbers could be a whole course in and of itself, so we won't go any deeper into the topic now. If you're interested in exploring this topic further, check out the ECMAScript specification (the specification on which JavaScript is based), and the Wikipedia page on binary-coded decimal numbers.

Note We're assuming you're using a modern browser on a modern computer, and Math.pow(2, 53) is based on the ability of JavaScript to represent 64-bit numbers, which modern browsers on modern computers can do. If, for some reason, you're on an older computer with an older browser, then your maximum number might be based on 32-bit numbers instead.

To Infinity (But Not Beyond)

You might remember from math class that if you divide by 0, you get infinity. When you begin programming, this can cause problems because if you try to represent infinity in some programming languages, well, let's just say your computer won't be too happy. JavaScript, however, is more than happy to represent infinity:

INTERACTIVE SESSION:

```
> var zero = 0;
undefined
> var crazy = 3/zero;
undefined
> crazy
Infinity
```

Instead of complaining when you divide by 0, JavaScript just returns the value `Infinity`. What is *Infinity* at least in JavaScript? (I suppose we may never know in the real world.)

INTERACTIVE SESSION:

```
> typeof crazy
"number"
```

In JavaScript, *Infinity* is a number (and so is *-Infinity*). Here's an experiment you can run if you like, but be prepared to close your browser window, because this is an experiment that will never end:

OBSERVE:

```
> var counter = 0;
undefined
> while (counter < crazy) {
      counter++;
      console.log(counter);
  }

1
2
3
... forever
```

So although you can represent *Infinity* in your programs, that doesn't mean you can ever reach it. You can test for it to prevent mistakes though:

INTERACTIVE SESSION:

```
> if (crazy == Infinity) { console.log("stop!"); }
stop!
< undefined
```

Not a Number

One final interesting number you should know about is *NaN*, or "Not a Number". Wait, something that means "Not a Number" is a number? Yes! Another oddity in the world of JavaScript. Give it a try:

INTERACTIVE SESSION:

```
> var invalid = parseInt("I'm not a number!");
undefined
> invalid
NaN
> typeof invalid
"number"
```

Here, we attempt to parse the string, `"I'm not a number"` into an integer. Clearly this will fail because there's nothing in the string, `"I'm not a number"` that resembles an integer. So what is the result in the variable `invalid`? Well, it's `NaN`, when we check the type of `invalid`, we see that it is indeed a `"number"`, even though the value is `NaN`.

You might think that you can test to see if a result is *NaN* like this:

INTERACTIVE SESSION:

```
> if (invalid == NaN) { console.log("invalid is not a number!"); }
undefined
```

It doesn't work though. Go ahead. Try it now. You won't see the console message `"invalid is not a number!"`

Instead, to test to see if a variable is not a number, you need to use the built-in function, `isNaN()`:

INTERACTIVE SESSION:

```
> if (isNaN(invalid)) { console.log("invalid is not a number!"); }
invalid is not a number!
< undefined
```

Be careful with `isNaN()` though. What do you expect if you write:

INTERACTIVE SESSION:

```
> isNaN("3")
false
```

You might have expected to see the result true (meaning that the string "3" is not a number), but we get false (meaning that the string "3" *is* a number). Why? Because `isNaN()` attempts to convert its argument to a number before it checks to see if it's not a number. In the case of "3", JavaScript succeeds. Think of this code as doing the equivalent of `parseInt("3")` and passing the result, 3, to `isNaN()`. This behavior is widely considered to be a bug in JavaScript and may be fixed in a future version. In the meantime, just make sure you know how `isNaN()` works so you can be prepared in case the value you pass to it *can* be converted to a number.

Objects

We've spent a lot of time in the world of primitives, so let's head on over to the world of objects for a while. At this point, you've probably had quite a bit of experience with objects, but let's do a quick review, to make sure you're set up for some of the more advanced object lessons to come.

Objects are collections of properties. Properties can be primitive values, other objects, or functions (which are called *methods* when they are inside an object). Let's take a look at an example of an object. We'll go ahead and create a simple HTML file to hold our object (it's easier than typing at the prompt in the console):

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var person = {
        name: "James T. Kirk",
        birth: 2233,
        isEgotistical: true,
        ship: {
            name: "USS Enterprise",
            number: "NCC-1701",
            commissioned: 2245
        },
        getInfo: function() {
            return this.name + " commands the " + this.ship.name;
        }
    };
  </script>
</head>
<body>
</body>
</html>
```

Save this file as *objects.html* in your work folder, and open it in your browser. You won't see anything in the page, so open up the console (and reload the page, just to be sure). Since we defined `person` as a global variable, we can use it in the console:

INTERACTIVE SESSION:

```
> person.getInfo()
"James T. Kirk commands the USS Enterprise"
```

The `person` object contains properties with primitive values and values that are other objects. We say that the `person.ship` object is *nested* inside the `person` object. You can nest objects within objects within objects and so on, as deep as you'd like to go (although there is a limit to how deep you can go, you're unlikely to hit it in a normal program), but keep in mind that the more nested objects you have, the more inefficient your object becomes. Also note that the most deeply nested object must have properties that are either primitive values or methods (in order for the nesting to stop).

Adding and Deleting Properties

One cool thing about JavaScript objects is that they are *dynamic*, that is, you can change the properties at any time by changing their values, or even by adding or deleting properties:

INTERACTIVE SESSION:

```
> person.title
undefined
> person.title = "Captain";
"Captain"
> person.title
"Captain"
> person
Object {name: "James T. Kirk", birth: 2233, isEgotistical: true, ship:
Object, getInfo: function}
birth: 2233
getInfo: function () {
isEgotistical: true
name: "James T. Kirk"
ship: Object
title: "Captain"
__proto__: Object
```

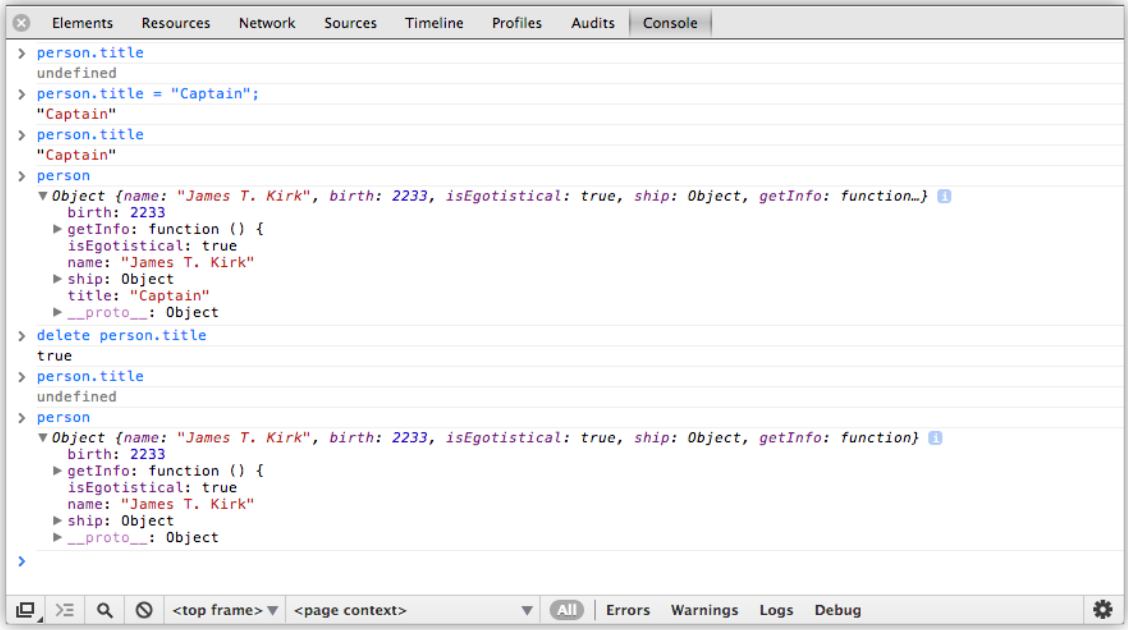
Here, we got the value of a property that doesn't exist in `person`, `person.title`. The result is `undefined`, which we'd expect. Next, we set the property `title` in person by defining it, and giving it the value `"Captain."` Now, we can get the value of the property using `person.title`. When we display the value of `person` in the console (just by typing the name of the object, and pressing *Enter*), we see that `title` has been added to the object, as if we'd had it there all along. (Note that we inspected the details of the `person` object by clicking on the arrow next to the object in Chrome, which exposes all of the details in the console.) Now, suppose you want to remove the `title` property:

INTERACTIVE SESSION:

```
> delete person.title
true
> person.title
undefined
> person
Object {name: "James T. Kirk", birth: 2233, isEgotistical: true, ship:
Object, getInfo: function}
birth: 2233
getInfo: function () {
isEgotistical: true
name: "James T. Kirk"
ship: Object
__proto__: Object
```

`delete` removes the *entire property*, not just the value. The property no longer exists in the object, so when we try to get its value, we get `undefined` again. When we inspect the object, we can see that the title property is gone.

Here's a screenshot of this console interaction in Chrome after loading *objects.html*:



What's the Type of an Object?

Are you wondering what the type of an object is? Go ahead and test using typeof in the console:

INTERACTIVE SESSION:

```
> typeof person
"object"
```

In this case, JavaScript returns the string "object" when we ask for the type of the object `person`. That's good.

Enumerating Object Properties

JavaScript has the capability to examine the properties of an object in the program itself. This is known as *type introspection*. Let's take a look at an example of that. Modify *objects.html* as shown.

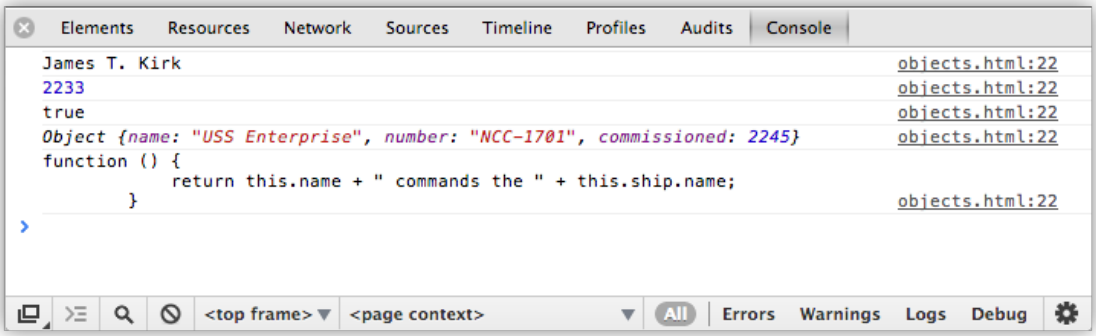
CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var person = {
        name: "James T. Kirk",
        birth: 2233,
        isEgotistical: true,
        ship: {
            name: "USS Enterprise",
            number: "NCC-1701",
            commissioned: 2245
        },
        getInfo: function() {
            return this.name + " commands the " + this.ship.name;
        }
    };

    for (var prop in person) {
        console.log(person[prop]);
    }
  </script>
</head>
<body>
</body>
```

```
</html>
```

Save these changes, and open or reload *objects.html* in your browser. You see each property of the object displayed in the console—here's what it looks like in Chrome:



Let's take a closer look at how we did this:

OBSERVE:

```
for (var prop in person) {
    console.log(person[prop]);
}
```

We used a *for* loop to loop through all the properties in the object, but instead of the traditional for loop you might be used to, we used a *for/in* loop (you may have used this in the previous phase of the Skills Ladder when accessing keys and values in Local Storage). In the for/in statement we declare a variable: `prop`. Each time through the loop, `prop` gets the property name of the next property in the object person. When we go through an object to access each of its properties, we call this *enumerating* an object's properties.

Then, inside the loop, we display the value of the object's property in the console. We use the *bracket notation* to access the object's property. You can try this at the console yourself to see how it works:

INTERACTIVE SESSION:

```
> person["name"]
"James T. Kirk"
```

To access an object's property value with bracket notation, you put the name of the property in quotation marks within brackets, next to the name of the object.

This notation is handy because it allows you to access the property of an object without knowing the name of the property in advance (look back at the for/in loop and see that we're using a *variable*, `prop`, as the property name within the loop).

When would you use this? Well, you might want to copy a property from one object to another, but only if the property does exist. You could use *bracket notation* to check to see if the property exists first. Or perhaps you are loading JSON data from a file using XHR (Ajax), and creating or modifying objects from the data. We'll see a couple of examples later in the course where the capability to enumerate an object's properties will come in handy.

Primitives That Act like Objects

Earlier we said that you can split JavaScript values into two groups: primitives and objects. This suggests that they are completely separate, and they are...for the most part. However, you should know that some primitives—specifically, numbers, strings and booleans— can *act* like objects sometimes.

Try this:

INTERACTIVE SESSION:

```
> var s = "I'm a string";
undefined
> s
"I'm a string"
> s.length
12
> s.substring(0, 3);
"I'm"
```

In the first line, we declare and initialize the variable `s` to be a string, "I'm a string." As you know, a string is a primitive. Yet we ask for the length of the string, `s.length`, and the first three letters of the string, `s.substring(0, 3)`, treating the string `s` as if it were an object. After all, only objects have properties, like `length`, and methods, like `substring()`, right? So, what's going on?

We have a primitive that's acting like an object! When you try to access properties and methods that act on a primitive, JavaScript converts the primitive to an object, uses a property or calls a method, and then converts it back to a primitive, all behind the scenes. In this example, the string `s` is changed to a *String* object *temporarily* so we can use the `length` property, and then changed back to a primitive. Then, it's converted to a String object so we can call the `substring()` method, and then changed back to a primitive again.

The same thing can happen with numbers and booleans:

INTERACTIVE SESSION:

```
> var num = 32;
undefined
> num
32
> num.toString()
"32"
> var b = true;
undefined
 > b.toString()
"true"
```

In practice, there are not many times you'll need your numbers and booleans to act like objects, except when you convert them to strings, which happens whenever you use `console.log()` like this:

INTERACTIVE SESSION:

```
> console.log("My num is " + num);
My num is 32
< undefined
```

Here, `num` is changed temporarily to a *Number* object, its `toString()` method is called, the result is concatenated with `"My num is",` and the result is displayed in the console (and `num` is converted back to a primitive). All of that happens behind the scenes so you don't have to worry about it.

Similar to built-in object types like *Array* and *Date* and *Math*, JavaScript has the built-in object types *Number*, *String* and *Boolean*. You'll rarely use these though, and you should never do this when you need just a simple primitive value, like 3:

INTERACTIVE SESSION:

```
> var num = new Number(3);
undefined
> num
Number {}
```

Why? Because JavaScript will always convert a primitive, like 3, to an object when it needs to, without you having to worry about it. So, primitives are converted to objects behind the scenes sometimes, but you'll probably never need to use those objects directly yourself.

JavaScript is Dynamically Typed

If you've had exposure to other languages, you might have run across languages in which you must declare the type of a variable when you create a new variable, like this:

OBSERVE:

```
int x = 3;
String myString = "test";
```

In these languages, once you declare a variable to be a certain type, that variable must always be that type. If you try to put a value of a different type into the variable, you'll get an error. For instance, you can't do something like this:

OBSERVE:

```
x = myString;
```

Here, we tried to set the variable `x` to a string, but we can't because `x` is declared to be an `int` (an integer number). This line of code will cause an error. These languages are known as "statically typed" languages. "Static" because the types of variables can't change.

Contrast this with JavaScript, which is a *dynamically typed language*. In JavaScript, you declare variables with no type, and you can change the types of the values in those variables at any time you want:

INTERACTIVE SESSION:

```
> var x = 3;
undefined
> var myString = "test";
undefined
> x = myString;
"test"
> x
"test"
```

So `x` starts out as a number, and ends up as a string. JavaScript has no problem with this.

You *can* change the type of a variable, but that doesn't mean you *should*. Why? Well, if you change the type of a variable in the middle of your program, you might forget you did and expect the variable to contain one kind of value, when in fact it might contain a different kind of value, which could cause bugs in your code.

Sometimes you'll take advantage of the fact that variables can contain any type, but most of the time, it's best to stick with one type for a given variable throughout your program.

In this lesson, you learned about primitives and objects that you might not have encountered before when programming in JavaScript. Understanding the types in JavaScript more deeply is important as you progress to more advanced programming. For instance, you might need to know whether to expect null or undefined if you're checking to see if a method succeeds or fails in creating a new object, or when to check to see if the result of a method is *NaN* if the user submits the wrong kind of data in a form.

Practice your new skills before moving on to the next lesson, where we'll continue to explore primitives and objects and how they behave when you start comparing them.

More about the material in this lesson

The Riverside JS Workshop would like to acknowledge the generosity of O'Reilly Media, Inc. for making this material available to us through the Creative Commons License. We would also like to acknowledge the great work of Elisabeth Robson who authored this content. She is one of our favorites. Elisabeth has written a number of *Head First* programming books for web developers. We recommend them highly. You can browse her work here.