

# Advanced JavaScript Essentials

## Lesson 1: **Introduction to Advanced JavaScript**

[Welcome to Advanced JavaScript](#)

[Accessing the Console](#)

[Using the Console](#)

[Good Programming Style Practices](#)

[Testing Code in the Console](#)

[Interacting Directly in the Console](#)

[Commenting Your Code](#)

[Quiz 1](#)

## Lesson 2: **Know Your Types**

[Know Your Types: Primitives and Objects](#)

[Primitives](#)

[Some Interesting Numbers](#)

[Objects](#)

[Enumerating Object Properties](#)

[Primitives That Act like Objects](#)

[JavaScript is Dynamically Typed](#)

[Quiz 1](#) [Project 1](#) [Project 2](#) [Project 3](#)

## Lesson 3: **Truthy, Falsey, and Equality**

[Truthy, Falsey, and Equality](#)

[Values That are Truthy or Falsey](#)

[Implied Typecasting](#)

[Testing Equality](#)

[Objects and Truthy-ness](#)

[Objects and Equality](#)

[Quiz 1](#) [Project 1](#) [Project 2](#)

## Lesson 4: **Constructing Objects**

[Constructing JavaScript Objects](#)

[Constructing an Object with a Constructor Function](#)

[Constructing an Object Using a Literal](#)

[Constructing an Object Using a Generic Object Constructor](#)

[So, What's the Best Way to Make an Object?](#)

[Initializing Values in Constructors](#)

[this](#)

[Constructing Array Objects](#)

[Quiz 1](#) [Project 1](#) [Project 2](#)

## Lesson 5: **Prototypes and Inheritance**

[Object-Oriented Programming in JavaScript](#)

[instanceof](#)

[Prototypes](#)

[Prototypes of Literal Objects](#)

[What is a Prototype Good For?](#)

[The Prototype Chain](#)

[Prototypal Inheritance](#)

[When are Prototype Objects Created?](#)

[hasOwnProperty](#)

[\\_\\_proto\\_\\_](#)

[Setting the Prototype Property to an Object Yourself](#)

[Quiz 1](#) [Quiz 2](#) [Project 1](#) [Project 2](#)

## Lesson 6: **Functions**

[JavaScript Functions](#)

[What is a Function?](#)

[Different Ways of Defining a Function](#)

[Functions as First Class Values](#)

[Anonymous Functions](#)

[Returning a Function from a Function](#)

[Functions as Callbacks](#)

[Calling Functions: Pass-by-Value](#)

[Return](#)

[Quiz 1](#) [Quiz 2](#) [Project 1](#) [Project 2](#)

## Lesson 7: **Scope**

[Scope](#)

[Variable Scope](#)

[Function Scope](#)

[Hoisting](#)

[Nested Functions](#)

[Lexical Scoping](#)

[Scope Chains](#)

[Inspecting the Scope Chain](#)

[Quiz 1](#) [Project 1](#)

## Lesson 8: **Invoking Functions**

[Invoking Functions](#)

[Different Ways to Invoke Functions](#)

[What Happens to this When You Invoke a Function](#)

[Nested Functions](#)

[When You Want to Control How this is Defined](#)

[call\(\) and apply\(\)](#)

[Function Arguments](#)

[The Four Ways to Invoke a Function](#)

[Quiz 1](#) [Project 1](#) [Project 2](#)

## Lesson 9: **Invocation Patterns**

[Invocation Patterns](#)

[Recursion](#)

[Why Use Recursion?](#)

[Chaining \(a la jQuery\)](#)

[Static vs. Instance Methods](#)

[Quiz 1](#) [Project 1](#) [Project 2](#) [Project 3](#)

## Lesson 10: **Encapsulation and APIs**

[Encapsulation and APIs](#)

[Privacy, Please](#)

[An Example](#)

[Private Variables](#)

[Private Functions](#)

[A Public Method](#)

[Accessing a Public Method from a Private Function](#)

[Encapsulation and APIs](#)

[Quiz 1](#) [Project 1](#) [Project 2](#)

## Lesson 11: **Closures**

[Closures](#)

[Making a Closure](#)

[What is a Closure?](#)

[Playing with Closures](#)

[Each Closure is Unique](#)

[Closures Might Not Always Act Like You Expect](#)

[Closures for Methods](#)

[Using Closures](#)

[Where We've Used Closures Before](#)

[Quiz 1](#) [Project 1](#)

## Lesson 12: **The Module Pattern**

[Module Pattern](#)

[IIFE or Immediately Invoked Function Expressions](#)

[The Module Pattern](#)

[Using the Module Pattern with JavaScript Libraries](#)

[A Shopping Basket Using the Module Pattern](#)

[Why Not Just Use an Object Constructor?](#)

[Quiz 1](#) [Project 1](#)

## Lesson 13: **The JavaScript Environment**

[JavaScript Runs in an Environment](#)

[The Core Language, and the Environment's Extensions](#)

[How the Browser Runs JavaScript Code](#)

[Including JavaScript in Your Page](#)

[The JavaScript Event Loop](#)

[The Event Queue](#)

[Asynchronous Programming](#)

[JavaScript in Environments Other Than the Browser](#)

[Quiz 1](#) [Project 1](#)

## Lesson 14: **ECMAScript 5.1**

[The ECMAScript Standard for JavaScript](#)

[Strict Mode](#)

[New Methods](#)

[Object Property Descriptors](#)

[Sealing and Freezing Objects](#)

[Creating Objects](#)

[Project 1](#) [Project 2](#)

---

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Introduction to Advanced JavaScript

---

Welcome to the O'Reilly School of Technology's (OST) Introduction to Advanced JavaScript course.

## Course Objectives

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

- create an object-oriented JavaScript program.
- structure your programs to make use of encapsulation where needed.
- write JavaScript using best coding practices.
- make use of patterns to structure your code.
- use and understand advanced techniques such as closures and recursion.
- obtain and utilize information about the environment in which JavaScript is running.

Before we begin programming, you need to learn a little about the programming environment you'll be using. This first lesson of the course will help you with that.

## Lesson Objectives

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

- use OST's Sandbox and learning tools.
- read about what to expect in the Advanced JavaScript course.
- review JavaScript basics.
- use the Developer Tools in your browser to access the JavaScript console, which we'll be using extensively in this course.
- use `console.log` to display messages in the console (for debugging).
- use good programming practices in your code.

---

## Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.

- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

## Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

**CODE TO TYPE:**

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

**INTERACTIVE SESSION:**

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type look like this.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

**OBSERVE:**

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

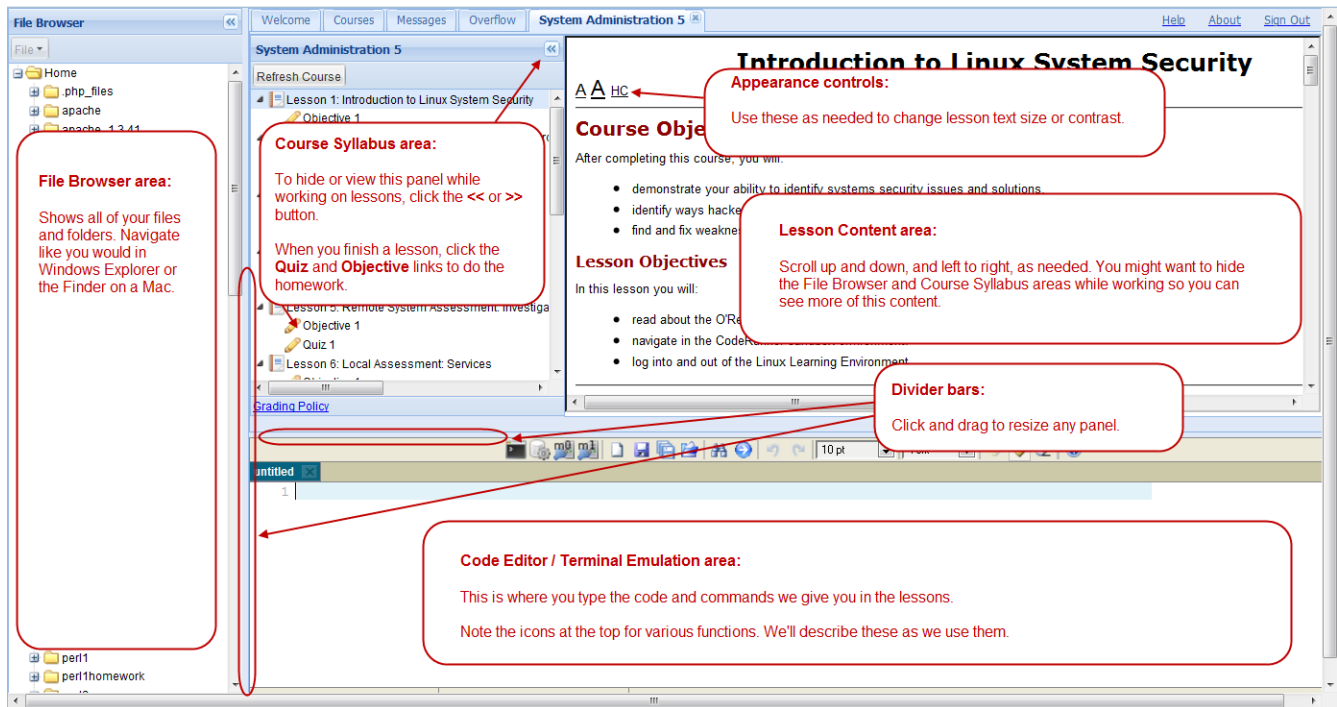
**Note** Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

**Tip** Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

**WARNING** Warnings provide information that can help prevent program crashes and data loss.

## The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

[File Management Demo](#)

[Code Editor Demo](#)

[Coursework Demo](#)

## Welcome to Advanced JavaScript

You can get started with JavaScript quickly. All you need is a text editor and a browser, and you can begin experimenting. When you learned the basics of JavaScript, you probably used it to modify web pages, and maybe to modify the style of your pages in response to user input. You've probably written event listeners to handle events like click events, and you've most likely used JavaScript to validate form data or add and remove elements from your page as users interact with it.

In this course, we'll focus on the JavaScript language itself. Since the primary place we use JavaScript is in the browser to create interactive web pages, we'll still build web page applications, but the focus will be on language features, rather than on web interfaces and the techniques we use to create web apps. The goal of this course is to take your understanding of JavaScript to a deeper level, from scripter to programmer. You'll learn how to leverage the powerful features of JavaScript in your programming, as well as how to avoid common mistakes.

### Accessing the Console

We'll make frequent use of the console to view the output we generate with `console.log`, as well as to inspect code. So, before we do anything else, let's make sure you're comfortable with the developer console, and you remember how to access and use it in each of the major browsers. Most end users never see the console because it's for developers who are creating, testing, and debugging code, so if you haven't had experience with the console before, don't worry, we'll get you up to speed quickly.

In this course, we'll show most examples using the Chrome browser console, because, as of this writing, it has the most functionality and is the easiest to use of the browser consoles. But you should become familiar with multiple browser consoles for testing and debugging your code.

## Note

Browsers are continually updated with new versions that include new versions of the console. So, while the basic functionality of the console will likely remain the same, your version of the console may look slightly different from what you see in this course. As you become familiar with the different browser consoles, you'll be able to figure out how to use each one.

Create a folder for your work. In the File Browser, right-click the **Home** folder and select **New folder...** (or click the **Home** folder and press **Ctrl+n**), type the new folder name **AdvJS**, and press **Enter**.

Now create a new HTML file, then add this code:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Getting Started </title>
  <meta charset="utf-8">
  <script src="basics.js"></script>
</head>
<body>
</body>
</html>
```



Save this as **basics.html** in your **/AdvJS** folder.


Now create a new JavaScript file and add this code:

### CODE TO TYPE:

```
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
  console.log("We have plenty left");
}
if (onSale || discount > 0) {
  console.log("On sale!");
} else {
  console.log("Full price");
}
```

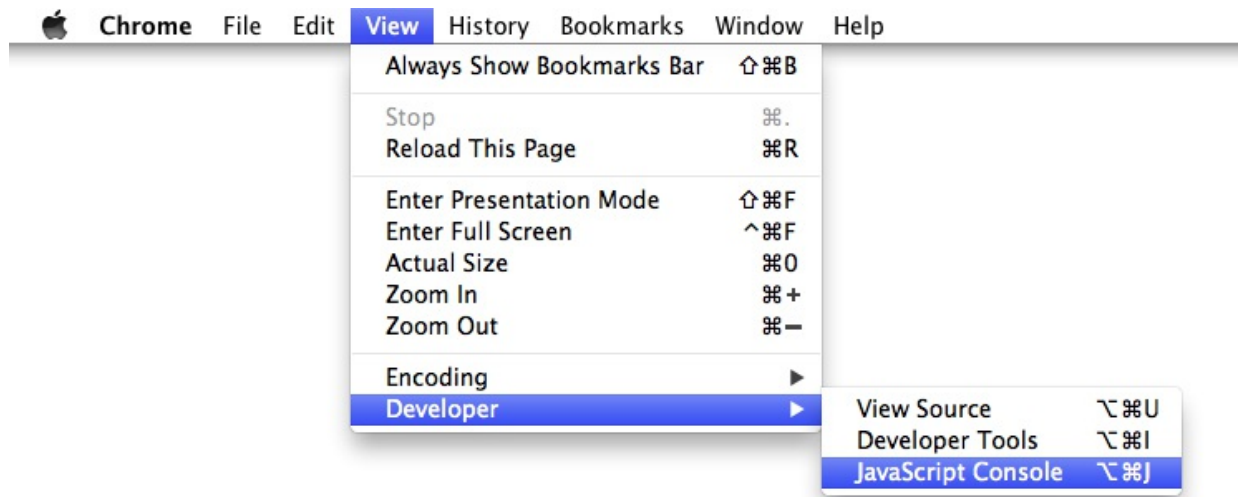


Save this as **basics.js** in your **/AdvJS** folder.

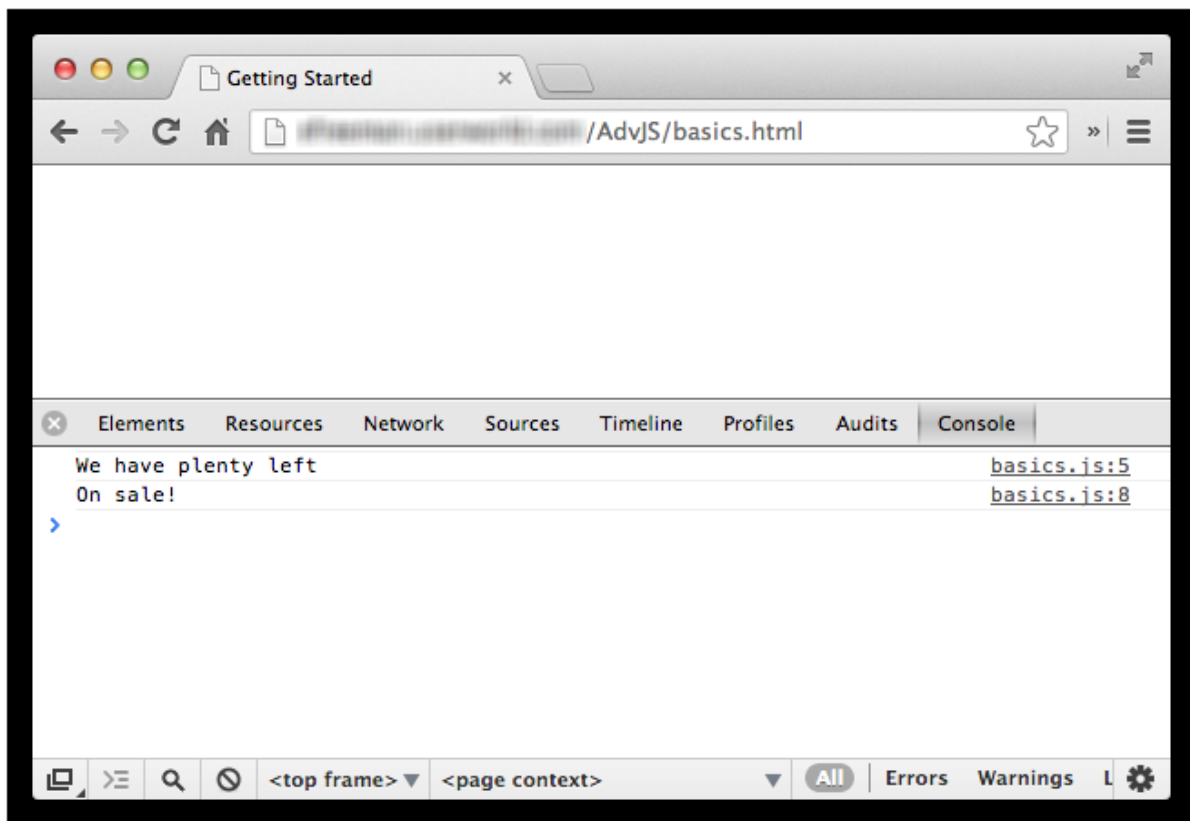
Now **Preview**  preview your **basics.html** file. You'll see an empty web page. To see the result of the JavaScript, open up your browser's console. To get instructions to access your console, click on the link to whichever browser you're using:

- [Chrome](#)
- [Safari](#)
- [Firefox](#)
- [Internet Explorer](#)

To access the console in Chrome, use the **View | Developer | JavaScript Console** menu:



Once you've enabled the console, you may need to reload the page to see the output. You'll see:

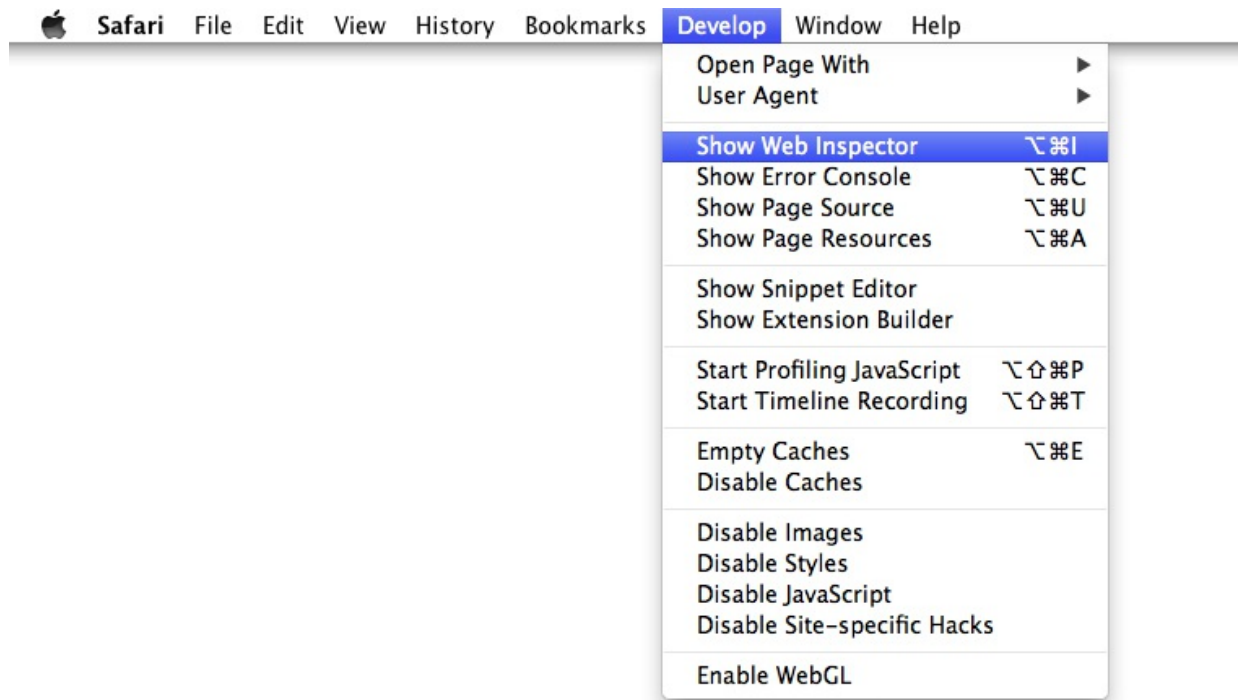


There are many parts to the console. You'll become more familiar with some of them as we work through the course.

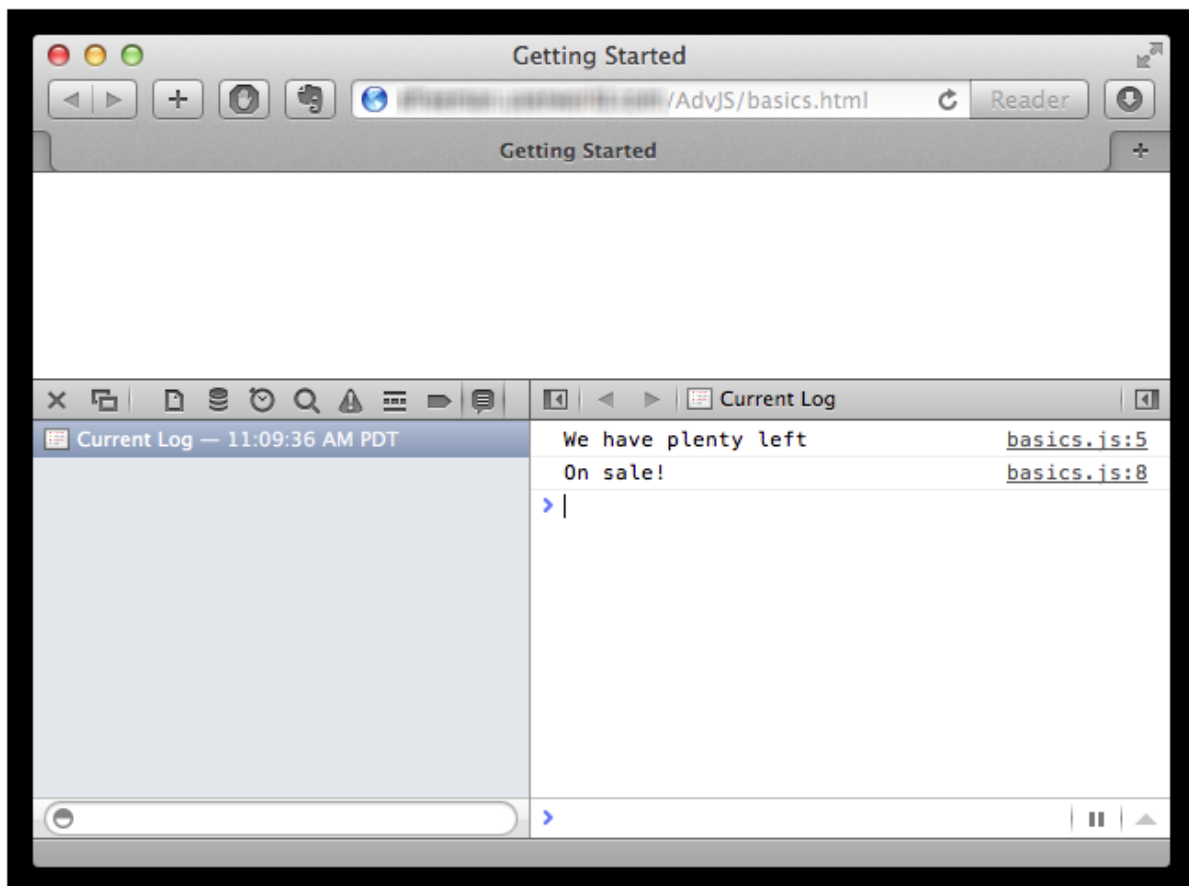
[Continue to the next step.](#)

To access the console in Safari, use the **Develop | Show Web Inspector** menu (if you don't have **Develop** enabled, you can enable it with **Preferences | Advanced | Show Develop menu in menu bar**):





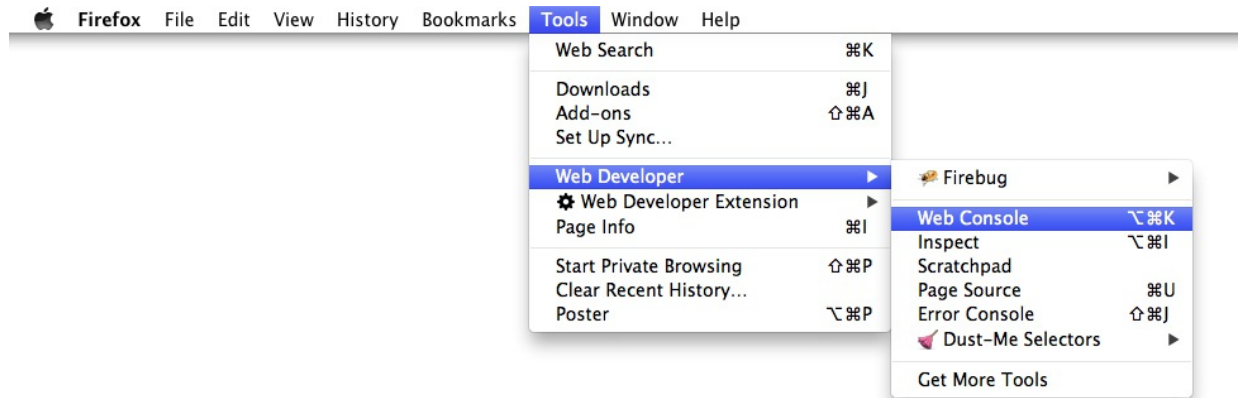
You might need to reload the page to see this output:



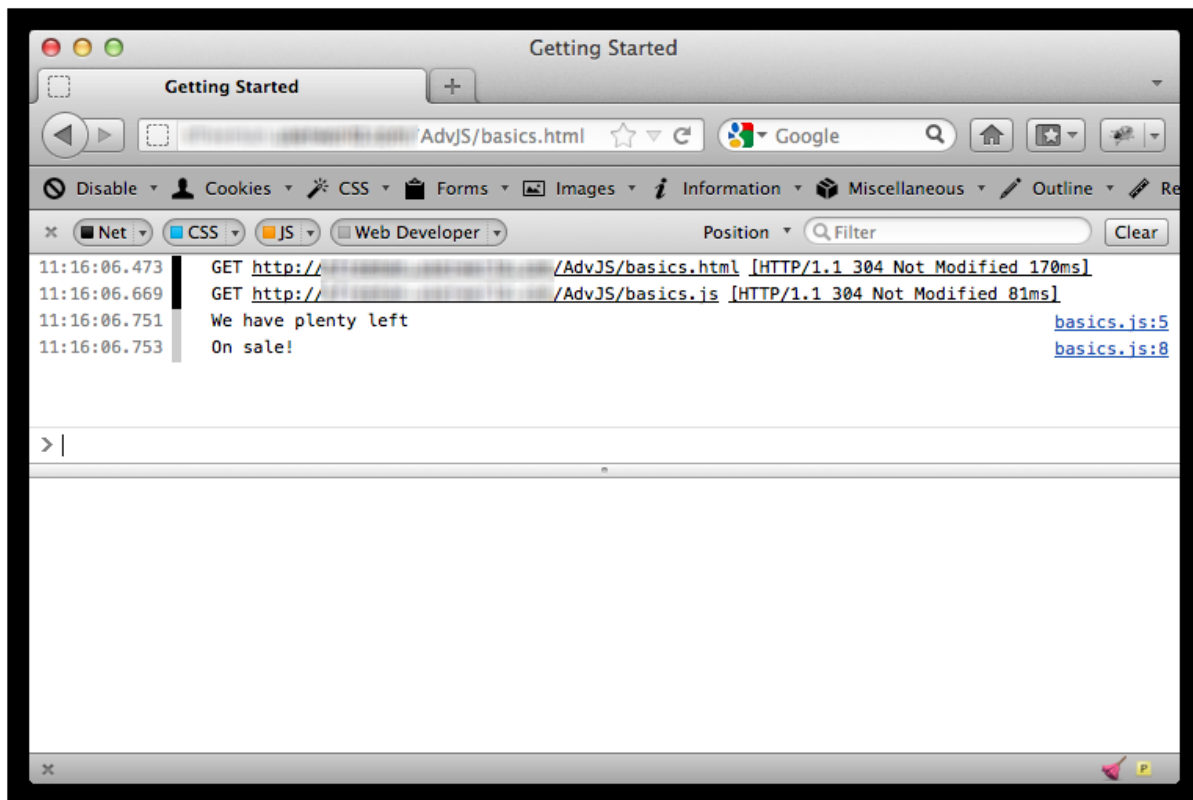
In order to see the output in the the Safari console, make sure you have selected the **Log** tab in the console, and you have the "Current log" at the *top* of the list on the left panel selected. (You may only see one log, but if you reload the page you could see previous logs. The current log is always at the top. This is where your most recent output from **console.log** will appear).

Continue to the next step.

To enable the Firefox console, use the **Tools > Web Developer > Web Console** menu and access the console:

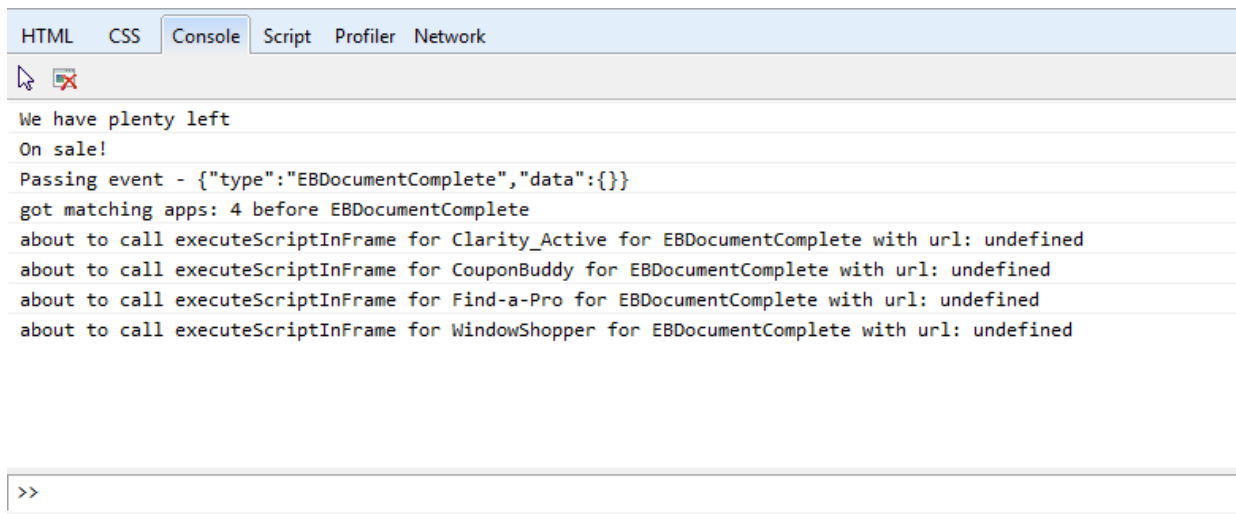


You may need to reload the page to see the output in the console. Note that in Firefox, the default location for the console is above the web page (unlike Chrome and Safari). You can click on the **Position** menu in the console and select "bottom" or "window" to change the position of the console. Choosing "window" will pop the console out into a separate window.



Continue to the next step.

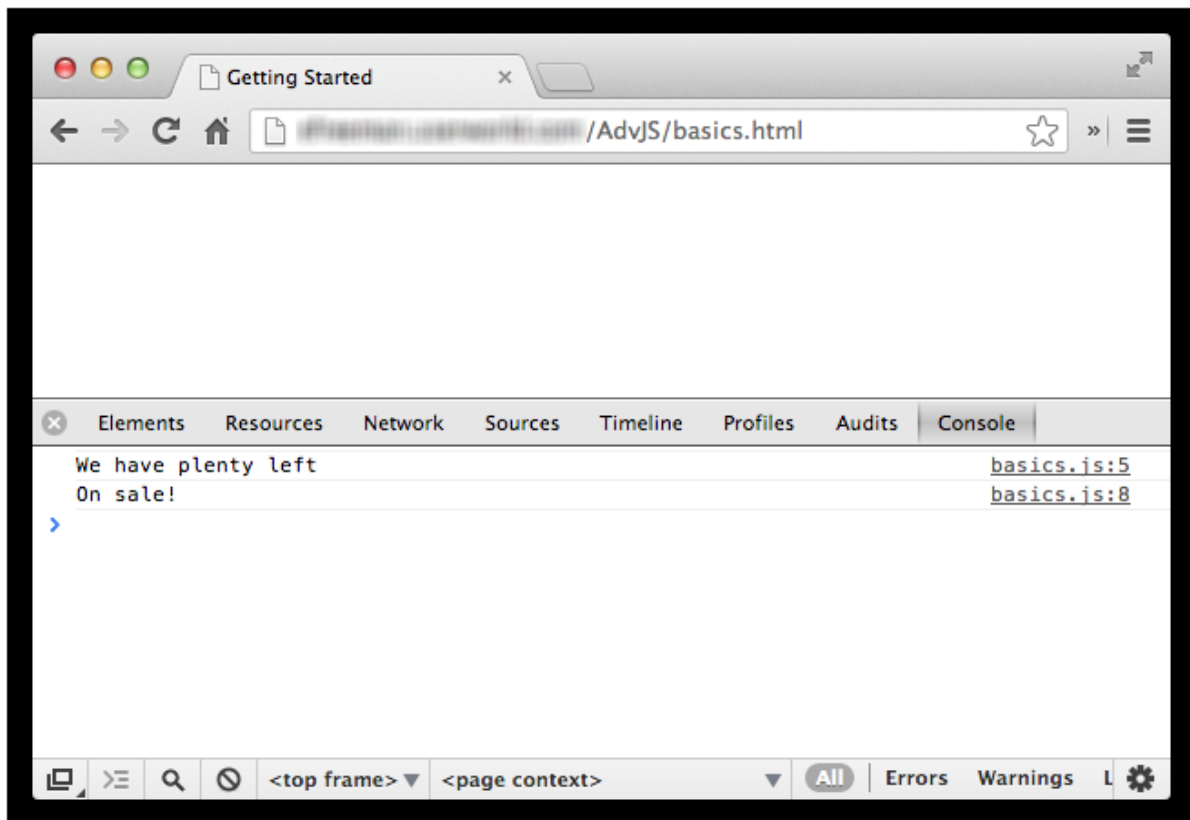
To enable the Internet Explorer console, select **Tools | F12 Developer Tools** and then select the **Console** tab.



You may need to reload the page to see the output in the console.

## Using the Console

Let's take a closer look at the Chrome console since that's the one we'll use in our examples throughout the course. Other browser consoles have the same basic functionality. We'll let you know when you need to use the Chrome console specifically to follow along.



There are several tabs across the top of the console, including the one we're on, **Console**. You'll use **Console** most often when looking at the results of `console.log()`, and debugging your code by checking to see if there are any JavaScript errors. In Safari, the equivalent is the **Log** tab shown in the screenshot; in Firefox, it's the default view you see when you access **Web Console** from the menu, and in IE, it's a tab labeled **Console**.

In Chrome, if you want to view the console into a separate window, click on the icon in the bottom left corner:



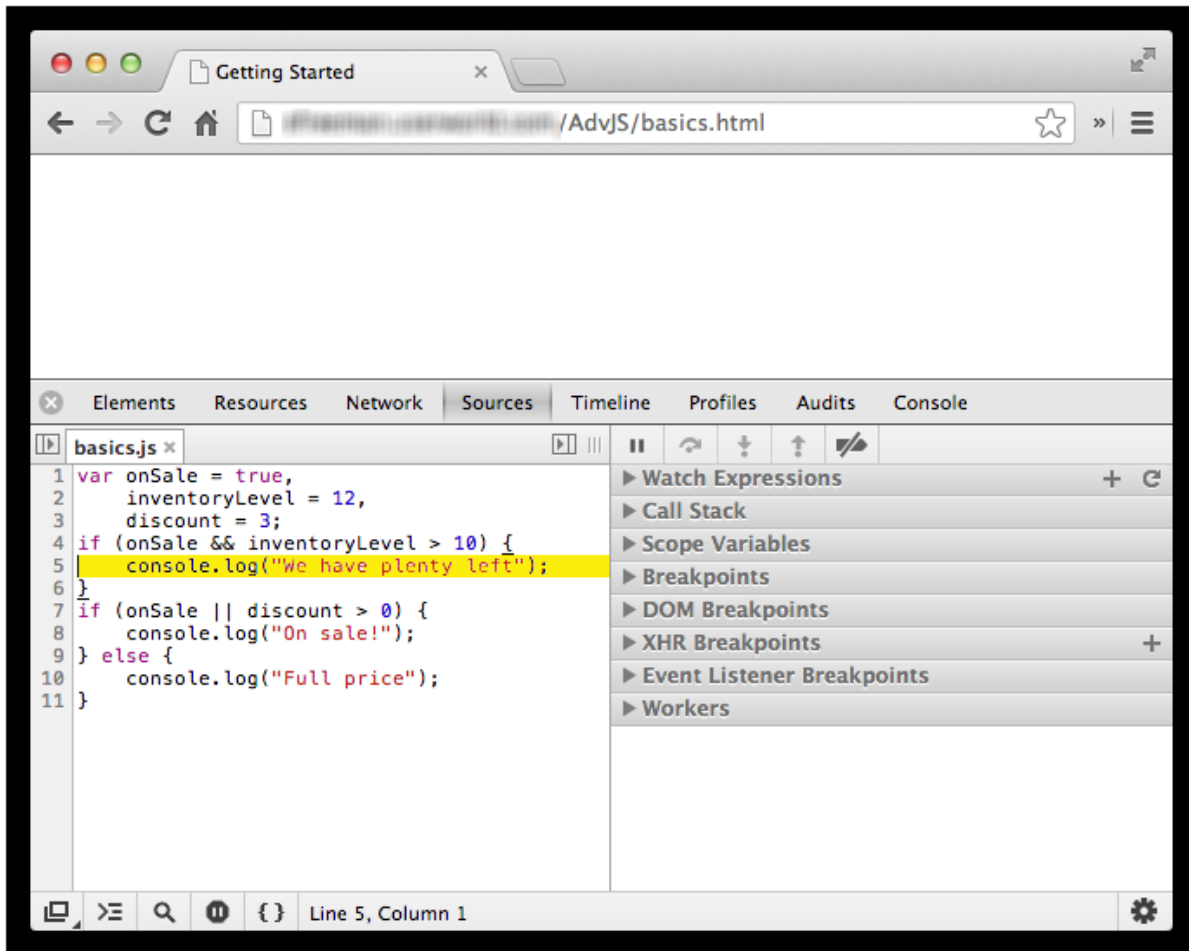
Click the **undock** icon now. This will create a separate window for the console. You can click the icon in the same location in that new window to dock it back to the original window. Give it a try.

You see two messages in the console that we created using **console.log** in our code:

```
OBSERVE:
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
  console.log("We have plenty left");
}
if (onSale || discount > 0) {
  console.log("On sale!");
} else {
  console.log("Full price");
}
```

The two **console.log()** messages shown in orange above create the output in the console. The file name in which the statements appear and the line numbers of the two statements are displayed in the console, next to the output.

In Chrome, try clicking one of the line numbers, like **basics.js:5**. This will open the **Sources** tab, and highlight that line in yellow temporarily:



This can help you track down potential bugs. There are whole lot of options on the right side of the "Sources" panel, including "Call Stack," "Scope variables," and "Breakpoints," all of which we'll use later as we get into some more advanced topics. Make sure that you have downloaded Chrome and have the latest version installed on your computer for testing. As of this writing, the current version is 26. Your version may be different, but that's okay because the basic functionality of the console is the same.

**Note**

Unfortunately, at this time, the other browsers don't come with the same tool installed by default. Go ahead and download Chrome and get familiar with the console tools, even if you typically use another browser to create web pages.

We'll explore more of the console later, but for now we'll take a closer look at the code.

## Good Programming Style Practices

Let's review some good practices for writing JavaScript programs:

**OBSERVE:**

```
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
  console.log("We have plenty left");
}
if (onSale || discount > 0) {
  console.log("On sale!");
} else {
  console.log("Full price");
}
```

**The variables we use are all declared at the top and given default values.** In general, it's good practice to declare your variables at the top of your file (or at the top of a function, if they are local variables), and to give them values. A variable that is *not* given a value is **undefined**. That's okay, but you need to be aware of which variables have values and which don't as you write your program. It's usually better practice to give your variables default values, rather than leave them undefined.

We've used the comma-separated style to declare the variables. It's also a good idea to put each variable declaration on a separate line. Still, you can always declare them with separate statements instead, like this:

```
OBSERVE:
var onSale = true;
var inventoryLevel = 12;
var discount = 3;
```

Either way is fine, so just do whatever you prefer.

Next, you'll see that we're using semicolons after *every* statement. While JavaScript doesn't require this currently, it's a good habit to develop. If you leave the semicolon off, JavaScript might interpret your statements and expressions in a way you're not anticipating. If you're in the habit of leaving off your semicolons, even occasionally, start putting them in after *every* statement. It will make debugging your code a whole lot easier. We suspect that future versions of JavaScript may require semicolons to delimit statements anyway, so you may as well get in the good habit now!

```
OBSERVE:
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
    console.log("We have plenty left");
}
if (onSale || discount > 0) {
    console.log("On sale!");
} else {
    console.log("Full price");
}
```

Another habit you should get into is using **curly braces** for every **if** (or **while** or **for**, and **such**) block of code, even if there's just one statement in the block. If you have only one statement in a block, technically you don't need to use the **{** and **}** characters to delimit the block. However, this is a bad habit because it can cause you to miss errors. Always use the curly braces!

Use plenty of white space. It's better to add more white space and format your code so it's easy to read, than to scrimp on space to make your code shorter. Readability is vital when working on larger, more complex programs, and it's always possible to "minify" your JavaScript later to take out the white space and make it more efficient to download.

**Note** There are plenty of "minification" programs out there that can minify your code for you such as [jscompress.com](http://jscompress.com) and [YUI compressor](http://YUI.compressor).

We'll cover other good programming practices and style suggestions throughout the course. We'll review them at the end of each lesson, so you'll get plenty of practice. There are also quite a few good style guides online if you want to explore this topic further (not all of them agree on everything, of course). We like the [Airbnb JavaScript Style Guide \(github\)](https://github.com/airbnb/javascript).

## Testing Code in the Console

Let's practice using the console to inspect values in our code. Update your file **basics.js** to define an object, **rect**, and then display it using `console.log`. Here, we define **rect** as an *object literal*; that is, an object we write as the value of the variable using the **{** and **}** characters to delimit the object. Remember, an object is just a collection of key value pairs. In this case **rect** has just two properties—width and height:

#### CODE TO TYPE:

```
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
  console.log("We have plenty left");
}
if (onSale || discount > 0) {
  console.log("On sale!");
} else {
  console.log("Full price");
}
var rect = {
  width: 100,
  height: 50
};
console.log(rect);
```

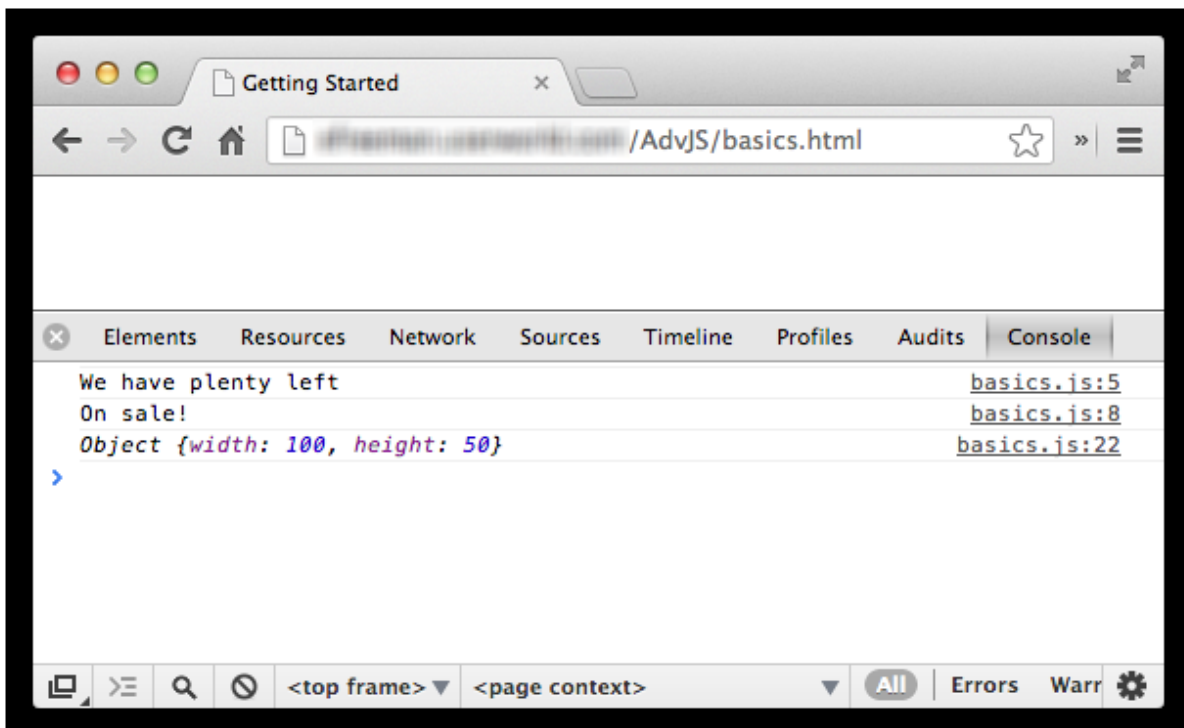


Save your changes (**basics.js**), and **Preview** preview your HTML file (**basics.html**), or simply reload the page if you still have it open in the browser. Make sure the browser console is open (you might have to reload the page to see the output). You'll see this output:

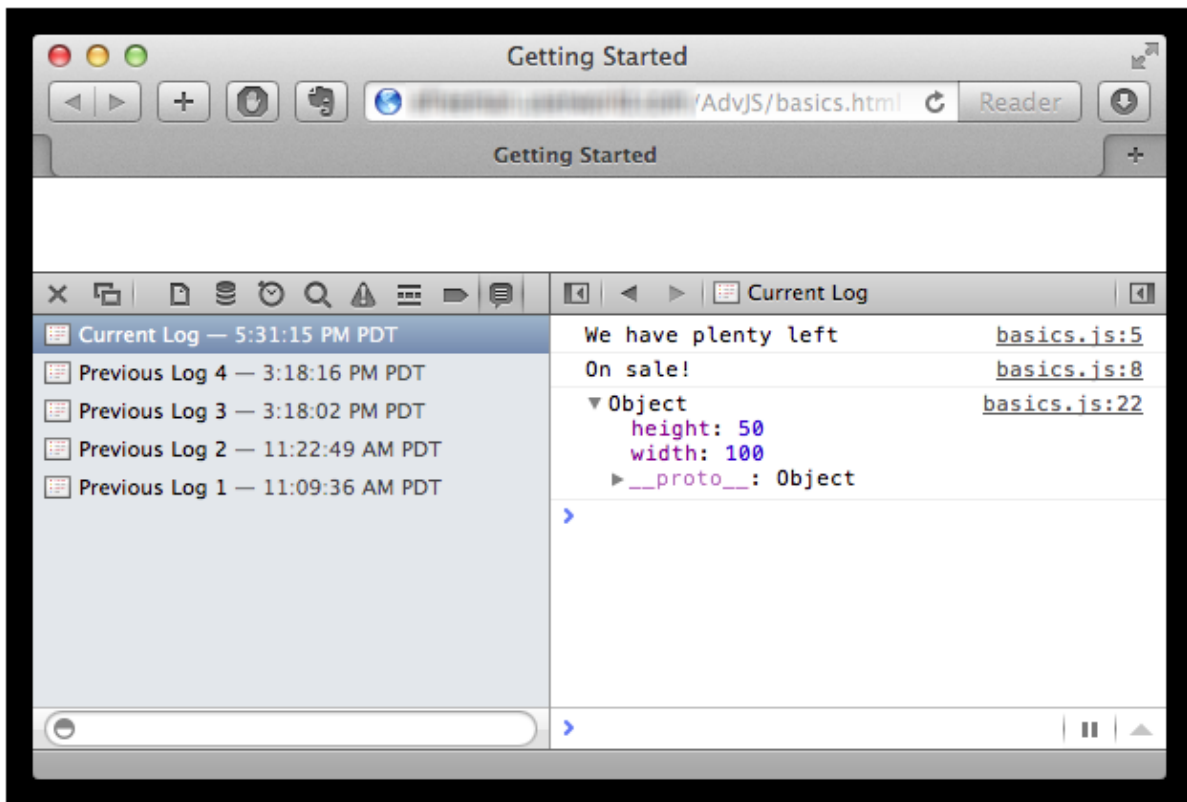
#### OBSERVE:

```
We have plenty left
On sale!
Object { width: 100, height: 50 }
```

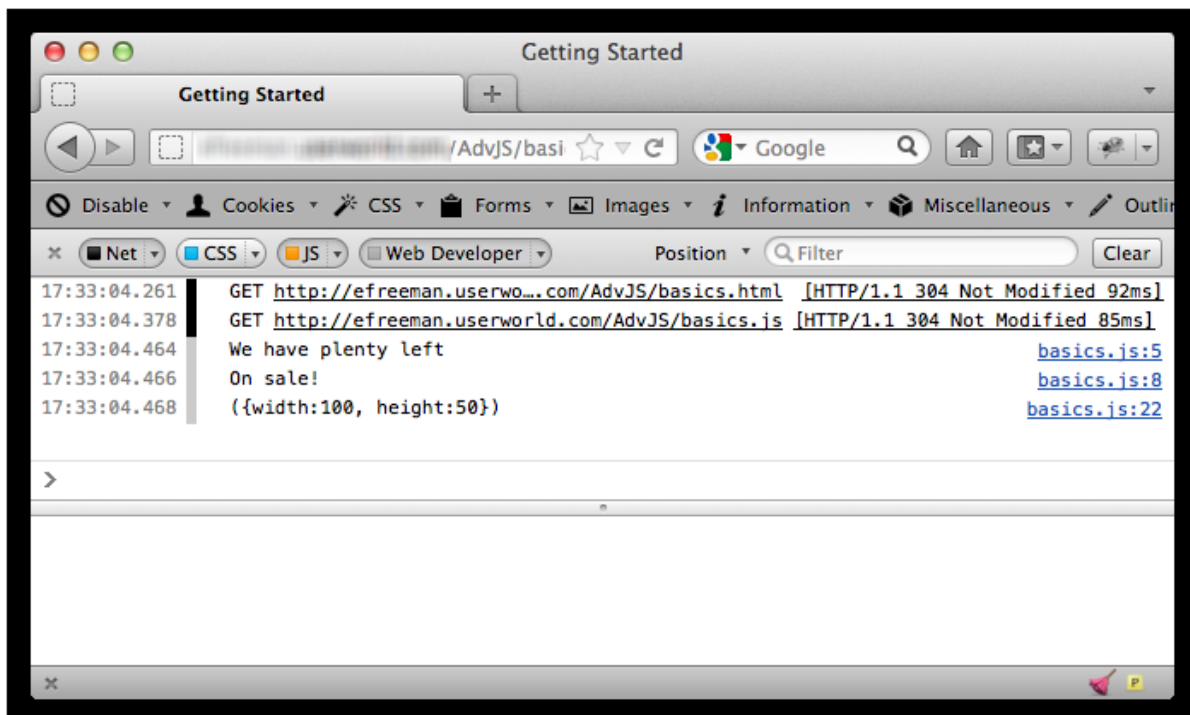
In Chrome, it will look like this:



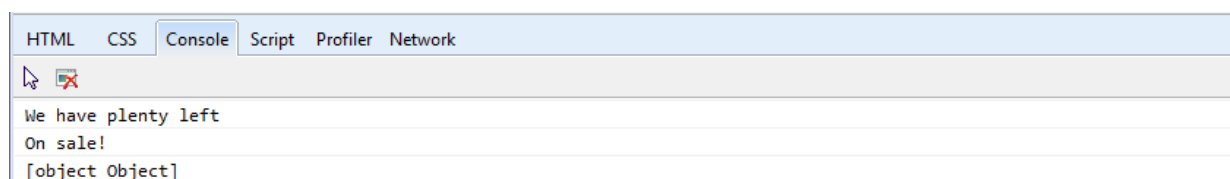
In Safari, it will look like this (make sure you click the little arrow next to the Object to see the object properties):



In Firefox, it will look like this:



In IE, it will look like this:





Notice that each is just a little different, but each console shows you the same basic information about the object, including the two property name/value pairs.

Now, let's make a small change to the code:

#### CODE TO TYPE:

```
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
  console.log("We have plenty left");
}
if (onSale || discount > 0) {
  console.log("On sale!");
} else {
  console.log("Full price");
}
var rect = {
  width: 100,
  height: 50
};
console.log(rect);
console.log("My object rect is: " + rect);
```



and **Preview**. In the console, you see:

#### OBSERVE:



```
Object { width: 100, height: 50 }
My object rect is: [object Object]
```

Why do you think this is? Instead of seeing the two properties that the object contains like we did before, we see just a string representation of the object, "[object Object]". That isn't really helpful. In our code, we're creating a string by concatenating the object, **rect**, with the string "My object rect is: ", before outputting the result to the console. Previously, we passed the object itself to **console.log()**, so the **console.log()** function displayed the object. Now we're passing a string to **console.log()**. When JavaScript converts the object to a string, we don't get a very useful display of the object.

Make one last change to your code:

#### CODE TO TYPE:

```
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
  console.log("We have plenty left");
}
if (onSale || discount > 0) {
  console.log("On sale!");
} else {
  console.log("Full price");
}
var rect = {
  width: 100,
  height: 50,
  toString: function() {
    return "Width: " + this.width + ", height: " + this.height;
  }
};
console.log(rect);
console.log("My object rect is: " + rect);
console.log("My object rect is: " + rect.toString());
```

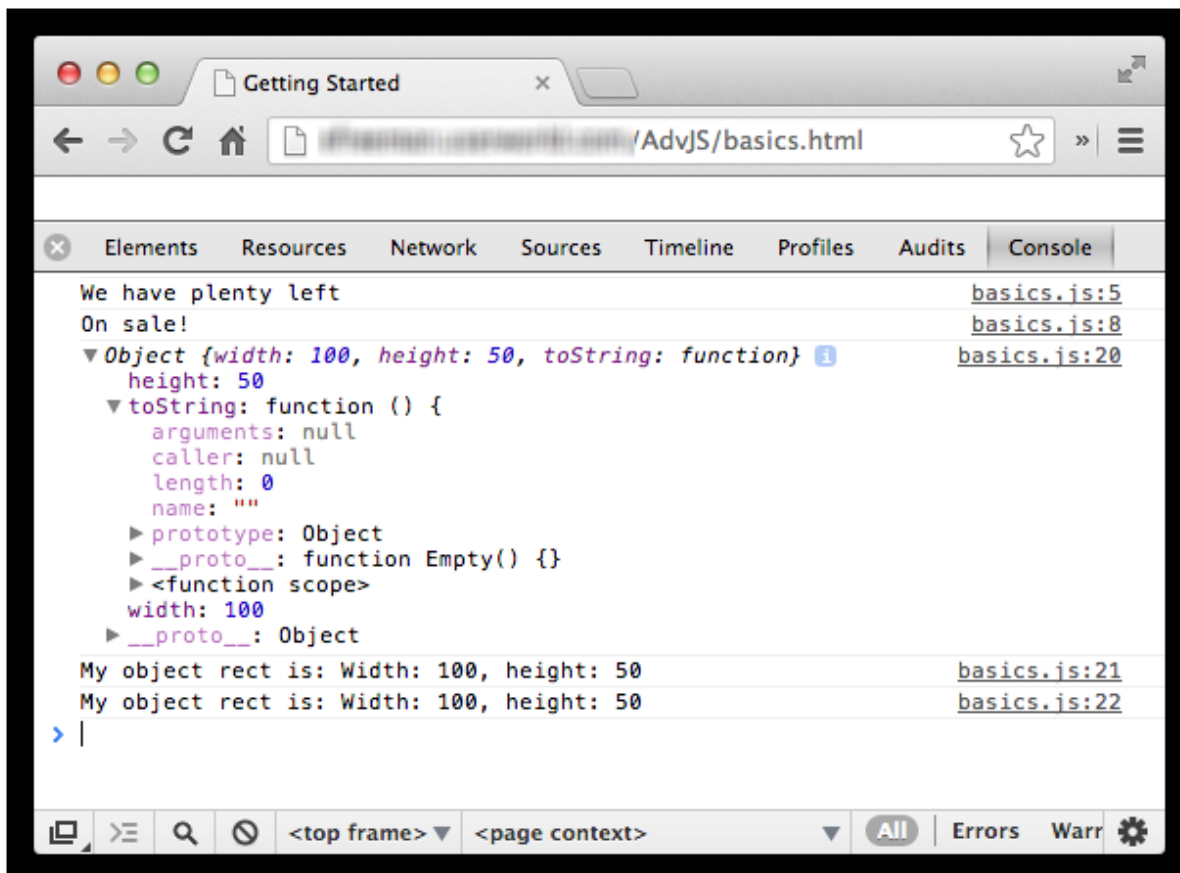
Don't forget the comma after the **height** property value! We added a method to this object as the third property value, so we need a comma between the second and third properties.  and .

```
OBSERVE:
Object { width: 100, height: 50, toString: function }
My object rect is: Width: 100, height: 50
My object rect is: Width: 100, height: 50
```

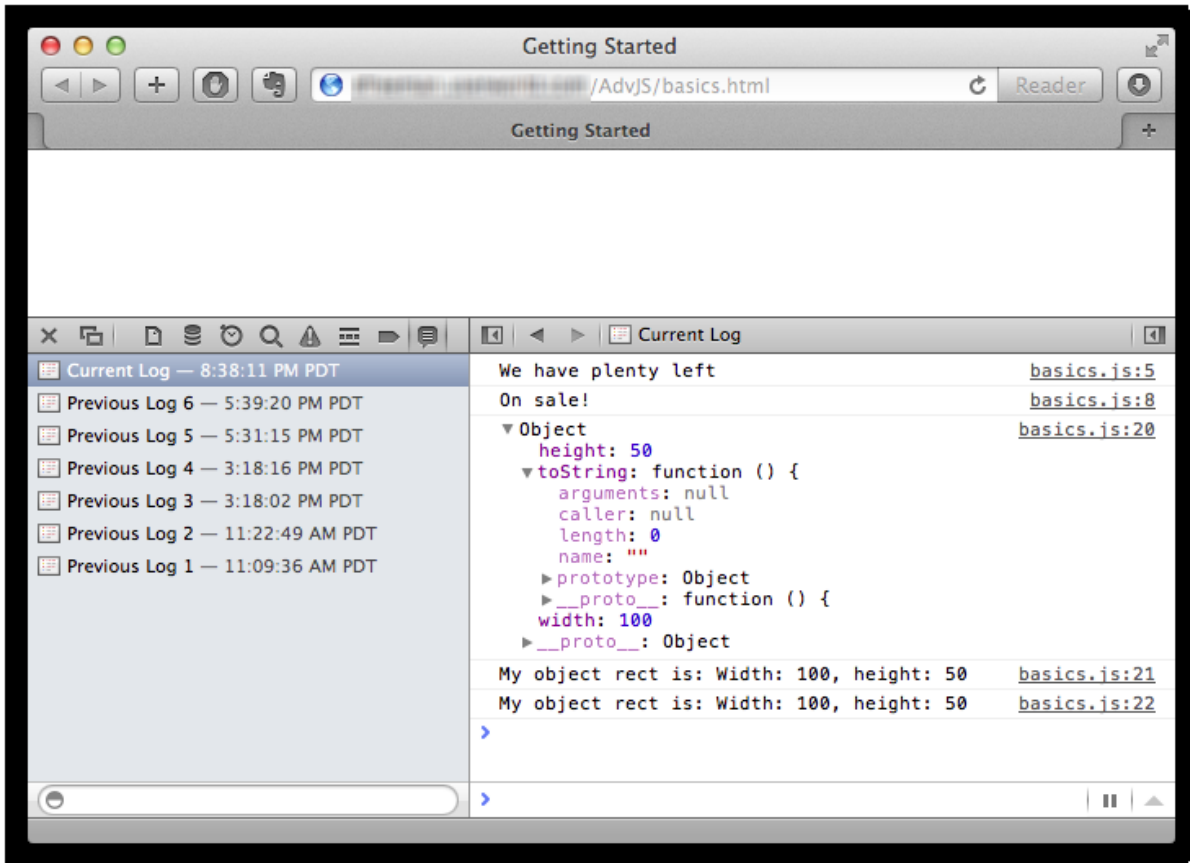
The **toString** method is now shown as part of the **rect** object in the output from **console.log(rect)**, which you can **see in the first line of output above**. The line of code we just added calls this method to display the width and height properties of the object, which you can see in **the third line of output above**. However, the **second line** that displayed "[object Object]" before, now displays the same as the third line. That's because JavaScript automatically calls the **toString()** method when converting an object to a string. If you implement the **toString()** method yourself in an object, then that's the method JavaScript will call. If you don't, then JavaScript calls the **toString()** method in the **Object** object, which is the parent object of all objects you create in JavaScript. The version of **toString()** that's implemented in the **Object** object doesn't do a very good job of creating a helpful string from the object, as you saw when "[object Object]" was displayed.

Don't worry about these details right now; we'll come back to all that later. For now just note the different ways that the console displays output, depending on how you call **console.log()** and the kind of value you pass this function.

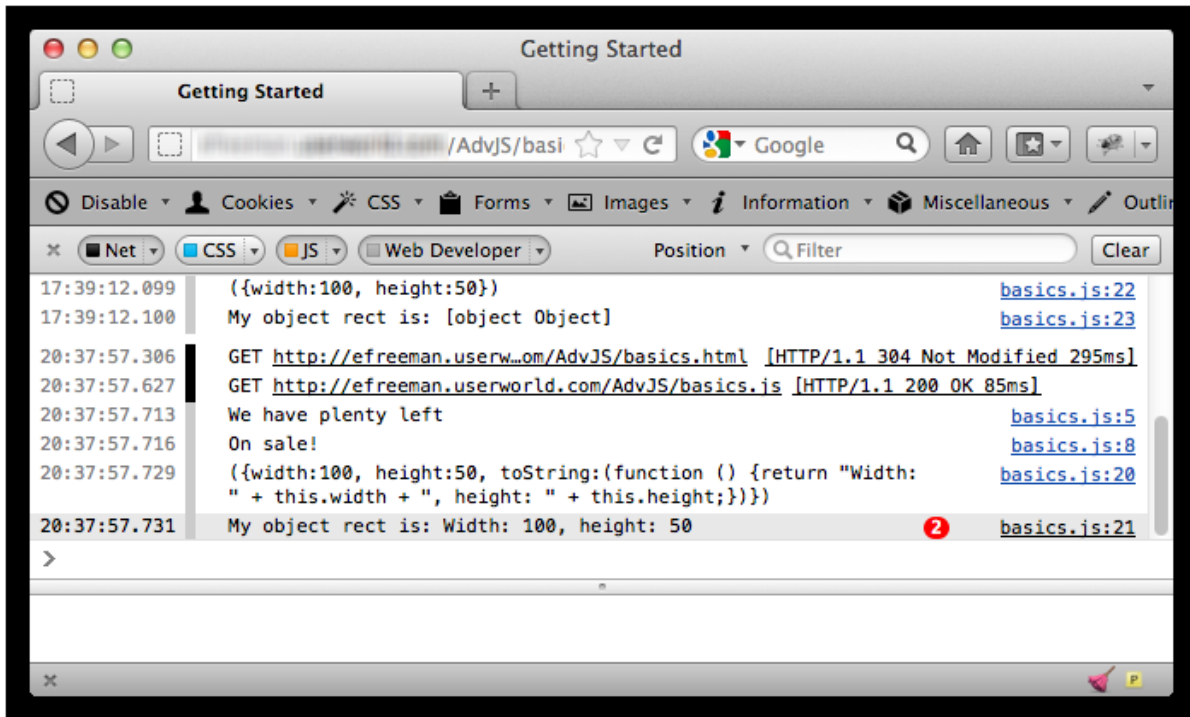
Here's the output in Chrome. In this screenshot, I've clicked on the line that shows the object properties (the third line in the output), which opens up the object to show more details, including lots of details about the **toString()** method. Again, don't worry about these details right now; we'll get to them later in the course, so you'll know what they mean at the end.



Safari (note that you can open up the object, and also the **toString()** method to see similar details in this console):



Firefox:



See the (2) next to the line of output? That means that the same line of output is displayed twice.

IE:

```
HTML CSS Console Script Profiler Network
We have plenty left
On sale!
Width: 100, height: 50
My object rect is: Width: 100, height: 50
My object rect is: Width: 100, height: 50
```

## Interacting Directly in the Console

So far, we've been using `console.log()` to display messages in the console, but you can interact directly with the console to display the values of variables, create new statements, and even modify your web page.

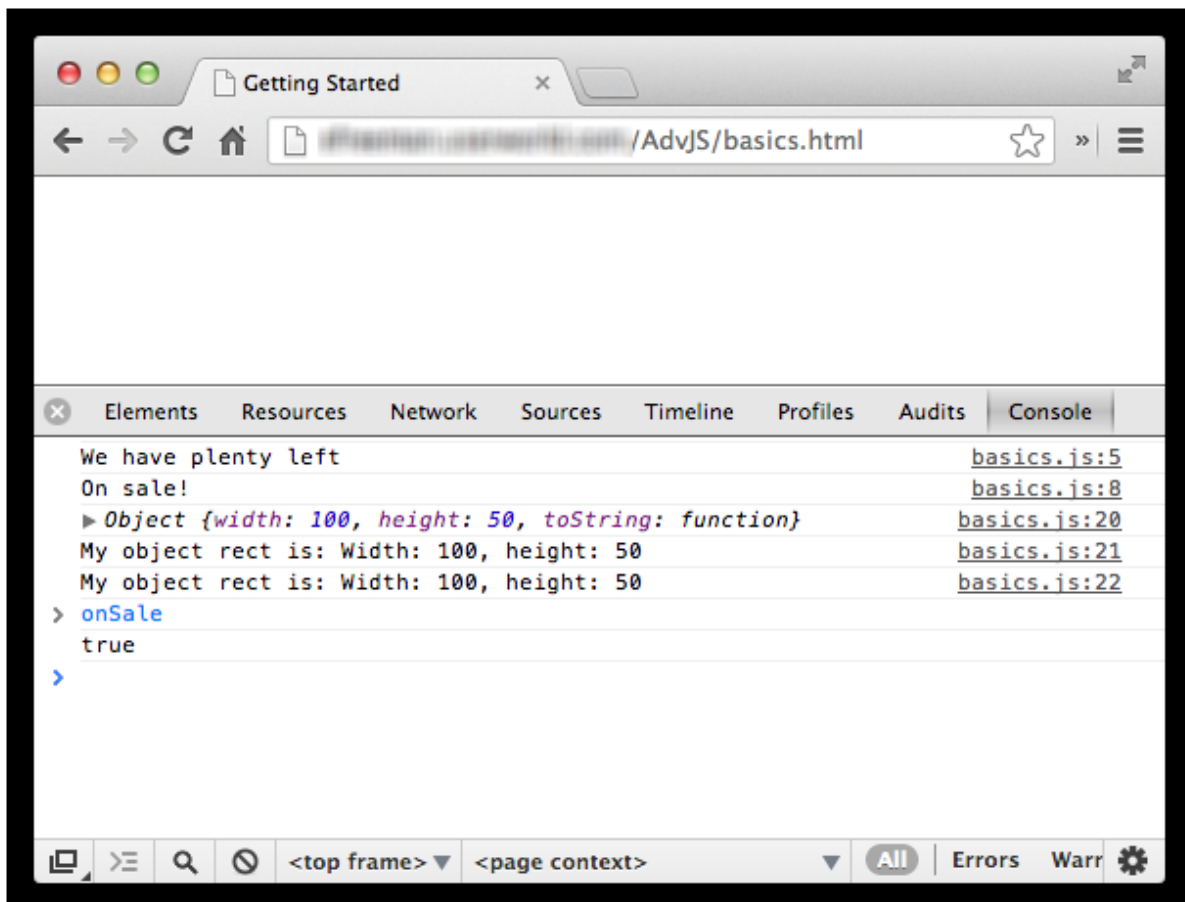
In your browser's console, you'll see a prompt which indicates where you can type your own JavaScript. Click in the console window next to the prompt, and type this:

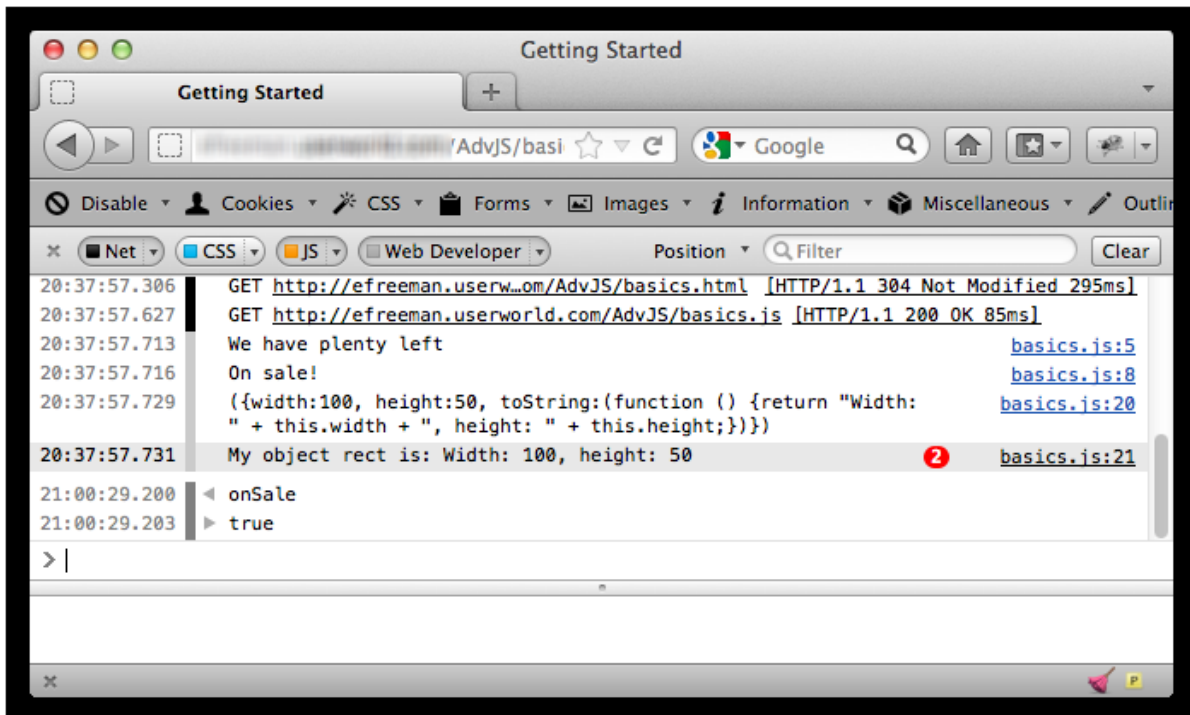
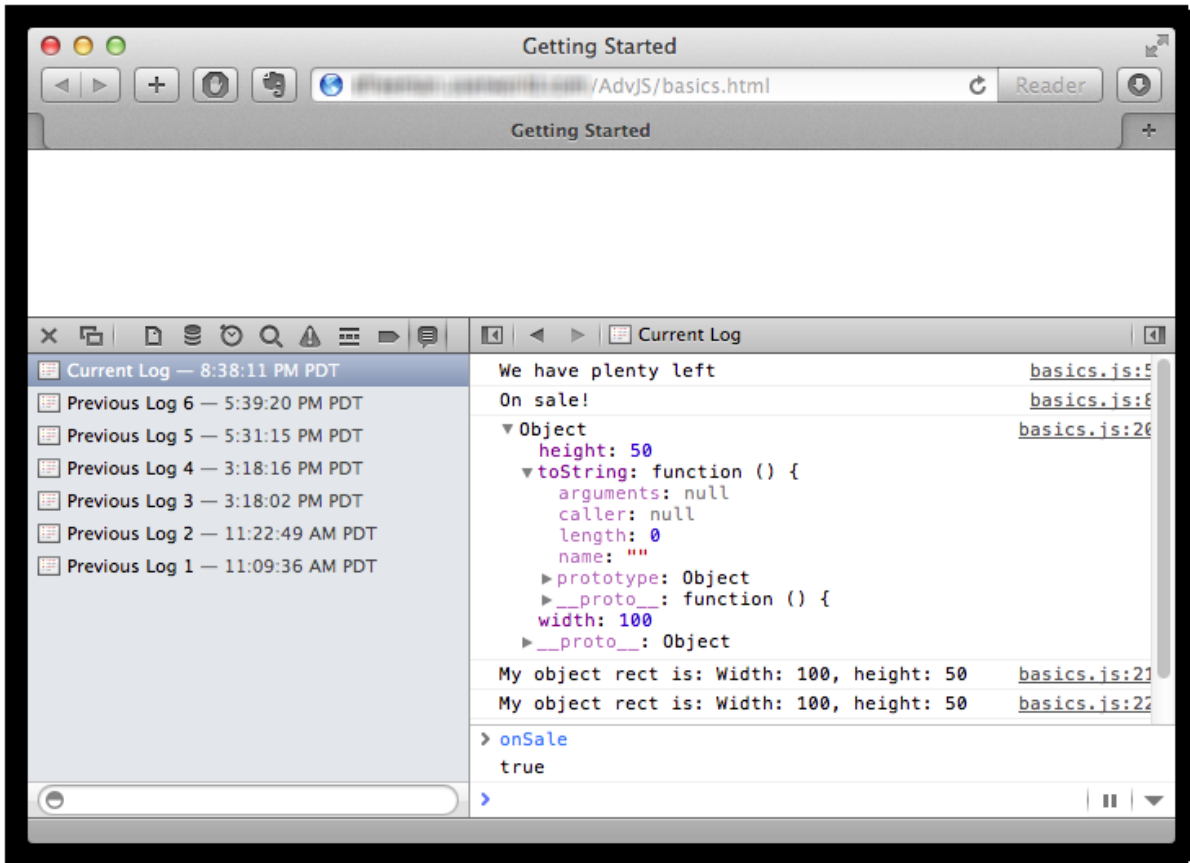
```
INTERACTIVE SESSION:
> onSale
true
```

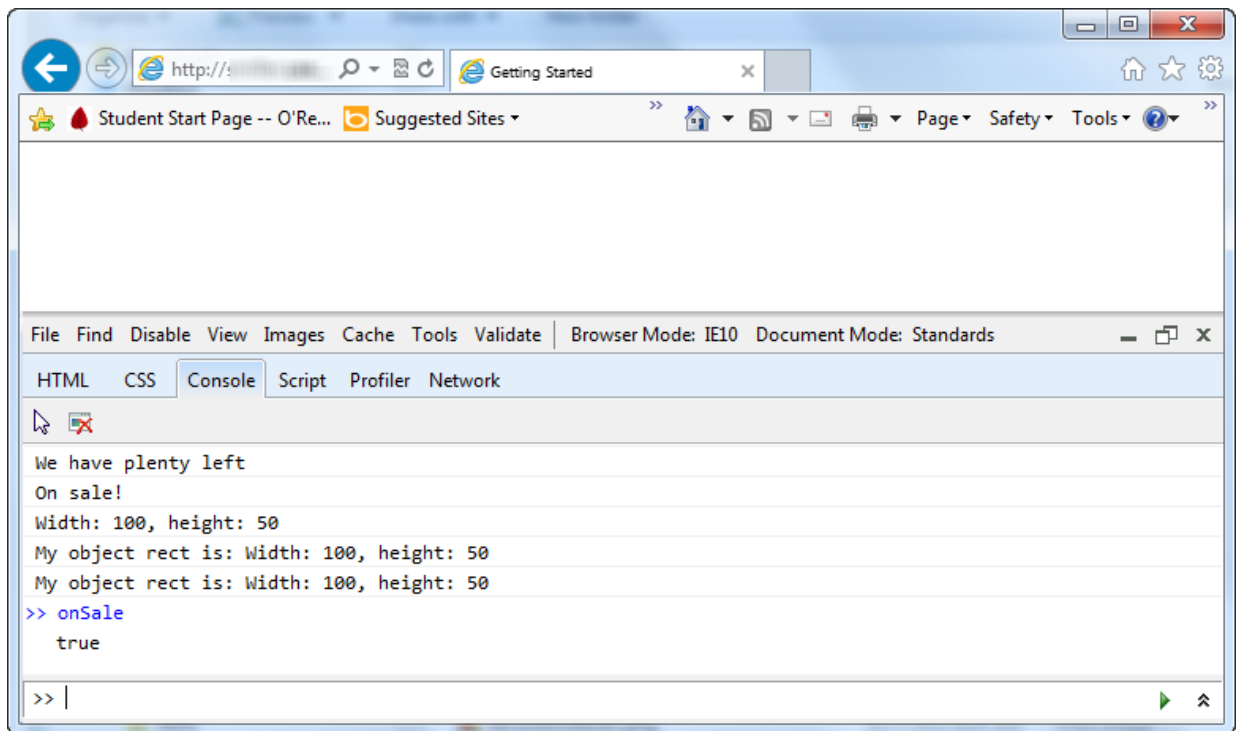
**Note** The ">" character is the prompt; you shouldn't type this part.

We typed the name of the global variable `onSale` and pressed **Enter**. JavaScript responded with the value of `onSale`.

Here's what the output looks like in the browsers Chrome, Safari, Firefox, and IE:







We typed the name of a global variable, **onSale** that we defined in the code that's currently loaded into this browser window. In response, the console provided the value of the variable, **true**. (Just think of **onSale** like an expression). Remember that you can only access *global variables* via the prompt, that is, variables defined at the global level in the currently loaded page. So far, all of the variables we've defined have been global. When we begin creating functions with local variables, you won't be able to access those via the prompt, but you can always display their values using **console.log()** in your code (we'll look at another way to inspect the values of local variables using the Chrome console later).

Try entering the names of the other global variables we've defined in this small program to see the output (**inventoryLevel**, **discount**, and **rect**). Try accessing the properties of the **rect** object (for example, **rect.width** and **rect.height**). Enter some other expressions like `2 + 3`, or "test." What happens? What happens if you enter the name of a variable that doesn't exist, like **testVar**?


You can also define new variables at the prompt. This can come in handy when you just want to try something out without creating a whole new file. For example, try this:

```
INTERACTIVE SESSION:
> var a = [1, 2, 3];
undefined
```

In this statement, you define a new variable, **a**, to have the value of an array with three values. You see the value **undefined**. That might be a bit confusing at first. It doesn't mean that the value of **a** is undefined, it just means that the value returned to the console as a result of executing this statement is undefined (that is, the value of the statement itself is undefined).

To verify that you've actually created a value for the array **a**, type **a** at the prompt:

```
INTERACTIVE SESSION:
> a
[1, 2, 3]
```

Now, let's write a loop to iterate over this array, right in the console. To do this, you'll either type the entire for loop on one line, or use **Ctrl+Enter** in Chrome and Safari, **Shift+Enter** in Firefox, and click the Multiline mode icon (  ) to create new lines in the console.

## INTERACTIVE SESSION:

```
> for (var i = 0; i < a.length; i++) {  
    console.log("a[" + i + "]: " + a[i]);  
}
```

If you forget to create a new line character and press **Enter** by mistake, you'll get an error and have to start over. Once you've got it typed in, press **Enter** to complete the code and you'll see this output:

## OBSERVE:

```
a[0]: 1  
a[1]: 2  
a[2]: 3
```

What happens if you reload the page? The value of **a** goes away. Variables that you add using the console are valid only for the current session, and go away when you reload or close the tab or window.

Experiment! Create some new variables and write some JavaScript statements in the console. Get familiar with using the console, especially in Chrome.

## Commenting Your Code

Of course, we can't end the lesson without mentioning comments. Comments are an important part of creating readable programs, especially when your programs get large and complex. As you probably know, there are two ways to comment code in JavaScript: `/* ... */` and `//`. Let's give these both a try:

## CODE TO TYPE:

```
/*  
var onSale = true,  
    inventoryLevel = 12,  
    discount = 3;  
if (onSale && inventoryLevel > 10) {  
    console.log("We have plenty left");  
}  
if (onSale || discount > 0) {  
    console.log("On sale!");  
} else {  
    console.log("Full price");  
}  
var rect = {  
    width: 100,  
    height: 50,  
    toString: function() {  
        return "Width: " + this.width + ", height: " + this.height;  
    }  
};  
console.log(rect);  
console.log("My object rect is: " + rect);  
console.log("My object rect is: " + rect.toString());  
*/  
  
//  
// This function computes the area of a circle  
//  
// @param {number} The radius of the circle  
// @return {number} The area of the circle  
//  
function computeArea(radius) {  
    return radius * radius * Math.PI;  
}  
  
console.log("Area is: " + computeArea(3));
```

Here, we used `/* ... */` to comment out large chunks of code. This is standard practice, as the `/*` and `*/` delimiters give you a quick and straightforward way to comment multiple lines, which can make testing and debugging easier.

We also used `//` to create several lines of comments above the new `computeArea()` function that we added to the code. We don't put all these comments in `/* ... */` though, because if we decide later to comment out the entire function, we can put the whole thing, including the heading comments, inside the `/*` and `*/` delimiters, which makes commenting large chunks of code (including multiple functions) a lot easier. Of course, we can use `//` to comment out single lines of code temporarily for debugging or providing comments within a function.

Finally, notice the style we've used to comment this function. We've provided a brief description for the function, as well as information about the parameter it expects, the **radius** of the circle, and the value it returns (which is the area of the circle).

We won't always comment the examples extensively, but get in the habit of commenting your project code. Your co-workers (and boss) will appreciate it, and so will you if (when) you need to go back and change your code later.

Take some time to experiment in the console. Add more `console.log()` statements in the example (or create your own example and experiment with that). Write statements directly in the console.

From here on, we'll show screen shots mostly from Chrome, but you can (and should) try the course examples in multiple browsers. We'll let you know when you need to use a specific browser console for testing.

Now that you've got the basics of using the console down, we'll dive right into the nitty gritty of JavaScript types in the next lesson.

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*



# Know Your 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 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 comparison 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 if it 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 `1.79769313486231570000000001e+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 as **objects.html** in your **/AdvJS** folder, and then **Preview** . 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, g  
etInfo: 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:

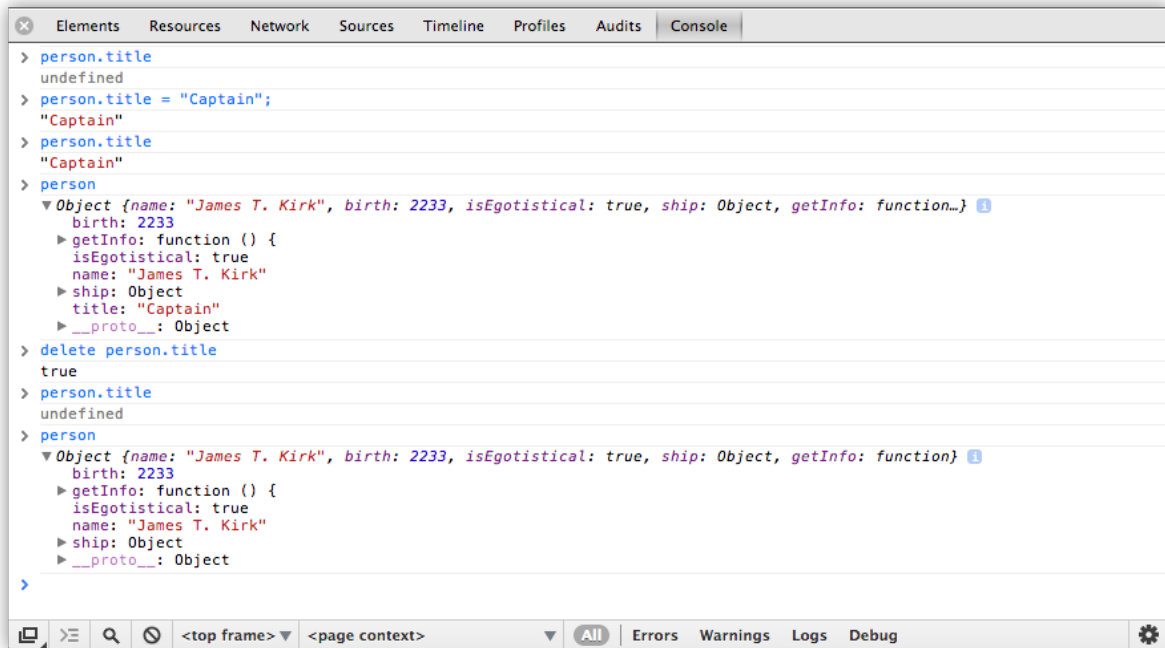
### INTERACTIVE SESSION:

```
> delete person.title  
true  
> person.title  
undefined  
> person  
Object {name: "James T. Kirk", birth: 2233, isEgotistical: true, ship: Object, g  
etInfo: 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**:



```
> 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
  }
> 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
  }
>
```

### 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:

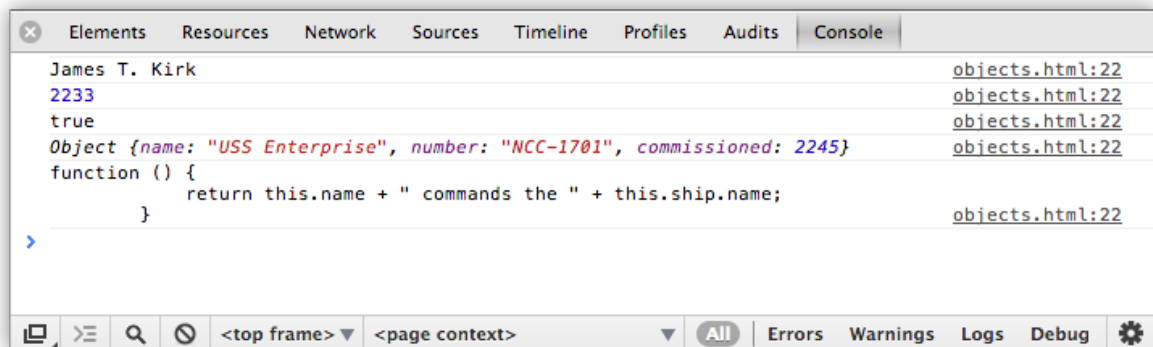
```
<!doctype html>
<html>
<head>
  <title> Objects </title>
  <meta charset="utf-8">
  <script>
    var person = {
      name: "James T. Kirk",
      birth: 2233,
      isEgotistical: true,
      ship: {
        name: "USS Enterprise",
        number: "NCC-1701",
        commissioned: 2245
      },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

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



and **Preview**

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 a previous course 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 with the quizzes and projects 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.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Truthy, Falsey, and Equality

## Lesson Objectives

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

- test for null and undefined.
- test values for truthiness.
- compare values using the strict equality operator.
- examine and compare the property names and the property values of objects.
- design appropriate conditional tests when using equality operators and typecasting.
- explore equality of objects.

You might think that testing for the equality of two values is simple and straightforward. Well actually, sometimes it is, and sometimes it's not. In JavaScript, when we compare two values, we get a result: true or false. However, determining the result of a comparison is not always as straightforward as you might think, because along with true and false, we also have *truthy* and *falsey* values. In addition, testing equality for primitives is different from testing equality for objects. In this lesson, you'll learn what's really happening behind the scenes when you compare values.

For the examples in this lesson, you're welcome to create an HTML file with a `<script>` or use the console. Either is fine. We'll show examples of both.

## Truthy, Falsey, and Equality

In JavaScript, we compare values all the time. For instance, you might do this:

OBSERVE:

```
var weather = "sunny";
if (weather == "sunny") {
  console.log("It's sunny today!");
} else {
  console.log("It must be rainy.");
}
```

We compare a string value with another string value using a *conditional expression*, `weather == "sunny"`, to see if they are equal, and the result of that conditional expression is either **true** or **false** (in this case, it's **true**). In an **if** statement, we use conditional expressions to determine whether to execute a block of code. If the expression in the parentheses results in **true**, the first block of code is executed; if it's not, the **else** block is executed (if there is one—if there isn't, execution just continues with the next line of code after the if statement).

We can do the same thing with values that are directly true or false like this:

OBSERVE:

```
var isItSunny = true;
if (isItSunny) {
  console.log("It's sunny today!");
} else {
  console.log("It must be rainy.");
}
```

Notice that here, we don't have to compare `isItSunny` to **true**, because we know that `isItSunny` is a boolean value. This means we can shorten the expression to `isItSunny`.

In many cases, we're working with expressions that are true or false, but sometimes we work with values that are *truthy* or *falsey*. What does this mean? It means that some values aren't directly true or false, but are interpreted by JavaScript to *mean* true or false in certain situations, like conditional expressions. Here's an example (before you try these in the console yourself, see if you can guess what you'll see as the result of each statement):

#### OBSERVE:

```
if (1 == 1) { console.log("1 really does equal 1"); }  
if (1) { console.log("1 is true"); }
```

Go ahead and try these statements in the console (remember that you might have to load an HTML page to access the console):

#### INTERACTIVE SESSION:

```
> if (1 == 1) { console.log("1 really does equal 1"); }  
1 really does equal 1  
< undefined  
> if (1) { console.log("1 is true"); }  
1 is true  
< undefined
```

The first statement is straightforward; we compare the value 1 with the value 1, so of course we expect them to be equal, and expect to see the console log message, "1 really does equal 1."

So what about the second statement? There, we test the value 1 to see if it's true or false, but 1 isn't either true or false, it's 1, right? Yet the result of this statement is that we *do* see the console log message "1 is true," which means that JavaScript must think that 1 is true. Hmm. That's perplexing.

What about this next example; what do you think you'll get?

#### OBSERVE:

```
if (0) { console.log("0 is true!"); }
```

Try it. This time we *don't* see the console log message, which means JavaScript must think that 0 is false:

#### INTERACTIVE SESSION:

```
> if (0) { console.log("0 is true!"); }  
undefined
```

Try one more experiment:

#### INTERACTIVE SESSION:

```
> if (-5) { console.log("-5 is true!"); }  
-5 is true!  
< undefined
```

That's interesting. JavaScript thinks that -5 is true!

So it turns out that numbers other than 0 are *truthy*, and 0 is *falsey*. We use those terms to indicate that even though -5 isn't true, it results in true in a conditional expression. Same with 0; even though 0 isn't false, it results in false in a conditional expression.

Experiment a bit on your own. For instance, is **NaN** truthy or falsey? What about **Infinity**?

## Values That are Truthy or Falsey

We need to find out which other values are truthy and falsey in JavaScript. Let's do some testing in the console to see how JavaScript treats values in truthy and falsey situations. We'll begin with **undefined**. Before you look at the example below, do you think **undefined** is truthy or falsey?

### INTERACTIVE SESSION:

```
> var myValue;
undefined
> if (myValue) { console.log("undefined is truthy!"); }
undefined
> if (!myValue) { console.log("undefined is falsey!"); }
undefined is falsey!
< undefined
```

Remember that **!** means NOT, so if **myValue** is false, then **!myValue** is true. So, **undefined** is falsey, because in the second expression, **myValue** resolves to **false**, and then we say "NOT false" with **!myValue**, which results in **true**, so we execute the if statement block to display the message "undefined is falsey!". Is that what you expected; that is, that **undefined** is falsey? Here's how this session looks in the Chrome console:

```
> var myValue;
undefined
> if (myValue) { console.log("undefined is truthy!"); }
undefined
> if (!myValue) { console.log("undefined is falsey!"); }
undefined is falsey!
< undefined
> |
```

What about **null**? Can you guess?

### INTERACTIVE SESSION:

```
> myValue = null;
null
> if (myValue) { console.log("null is truthy!"); }
undefined
> if (!myValue) { console.log("null is falsey!"); }
null is falsey!
< undefined
```

So, **null** is also falsey. It kind of makes sense that if **undefined** is falsey, then **null** would also be falsey, right?

Okay, we've looked at numbers, undefined and null. What about strings?:

### INTERACTIVE SESSION:

```
> var myString = "a string";
undefined
> if (myString) { console.log("myString is truthy!"); }
myString is truthy!
< undefined
> if (!myString) { console.log("myString is falsey!"); }
undefined
```

In this case, we see the string "myString is true" which means that **myString** is truthy. How about if we set **myString** to the empty string, **""**. Now do you think **myString** will be truthy or falsey?



## INTERACTIVE SESSION:

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

So a string with characters is truthy, but an empty string is falsey. Experiment a bit. What if **myString** is a string with one space in it, like this: " "?

### Shortcuts using truthy and falsey results

Knowing that **undefined**, **null**, and **""** are all falsey values, can you think of a good way to shorten the code below?

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Truthy, Falsey, Equality </title>
  <meta charset="utf-8">
  <script>
    var myString = prompt("Enter a string");
    if (myString == null || myString == undefined || myString == "") {
      console.log("Please enter a non-empty string!");
    } else {
      console.log("Thanks for entering the string '" + myString + "'");
    }
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **stringTest.html**, and **Preview** . Open the console (you might need to reload the page to see the output in the console). Enter a string and note the message you see. Try entering different values at the prompt, like **null** (just click **OK**), **""**, and **"test"**, for instance.

Now that you know about truthy and falsey values, you can shorten this code:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Truthy, Falsey, Equality </title>
  <meta charset="utf-8">
  <script>
    var myString = prompt("Enter a string");
    if (myString == null || myString == undefined || myString == "" !myString) {
      console.log("Please enter a non-empty string!");
    } else {
      console.log("Thanks for entering the string '" + myString + "'");
    }
  </script>
</head>
<body>
</body>
</html>
```



and **Preview** again (or reload if you still have the page open). You get the same behavior as before, but with a much shorter conditional expression. Again, try entering a few different values to make sure this works as you expect. Try replacing the prompt for **myString** to **undefined**, **null**, the empty string, or something else.

In cases where you need to test to make sure that a variable has a truthy value, but you don't care what that value is exactly, you can use this type of shortcut to save yourself some typing.

Be careful though. In cases where you need to test for a specific value, you also need to understand *implied typecasting*. We'll talk about next.

## Implied Typecasting

Does 88 equal "88?" That's an interesting question. Let's see:

#### INTERACTIVE SESSION:

```
> var myNum = 88;
undefined
> var myString = "88";
undefined
> if (myNum == myString) {
  console.log("My number is equal to my string!");
}
My number is equal to my string!
< undefined
```

JavaScript converts the string "88" into a number *before* doing the comparison between `myNum` and `myString`, so the comparison actually happens between 88 and 88. Of course those values are equal, so the result is true, and we see the console log message, "My number is equal to my string!"

This process of converting a string to number before doing a comparison or another operation is called "implied typecasting" (also known as type coercion or type conversion). JavaScript does implied typecasting as needed. For instance, whenever you do something like this:

### INTERACTIVE SESSION:

```
> var age = 29;
undefined
> var output = "My age is " + age;
undefined
> output
"My age is 29"
```

Here you are using JavaScript's ability to convert the variable **age** from a number to a string automatically, so it can be concatenated with the string "My age is."

When converting strings and numbers, be careful because the result may not always be what you expect. You know that sometimes JavaScript converts a string to a number, like when we compare 88 and "88," and you know that sometimes JavaScript converts a number to a string, like when you want to concatenate a number to a string. So what do you think the result will be when we try to add a string that *contains* a number to a number?

### INTERACTIVE SESSION:

```
> var x = 4;
undefined
> x = x + "4";
"44"
> x
"44"
```

You might've expected to get 8.

Let's try another experiment:

### INTERACTIVE SESSION:

```
> var myString = "test";
undefined
> if (myString) {
  console.log("'test' is truthy");
} else {
  console.log("'test' is falsey");
}
'test' is truthy
< undefined
> if (myString == true) {
  console.log("'test' is true");
} else {
  console.log("'test' is false");
}
'test' is false
< undefined
```

Notice that in the first if statement, **if (myString) ...**, we rely on the truthy-ness or falsey-ness of **myString** to result in true or false to determine the flow of execution. However, in the second if statement, **if (myString == true) ...**, we compare the value of **myString** with the boolean **true**, explicitly. JavaScript doesn't do implied typecasting and conversion of **myString** to true or false here.

As you can probably tell, it's a bit tricky to know for sure in every case exactly how JavaScript will (or won't) typecast and convert a value for comparison. One way that you can be more confident that you'll get the result you expect is to use the **strict equality operator ===**, in place of the **equality operator, ==**.

For more information about how JavaScript performs type conversions, see the [ECMAScript specification](#).

## Testing Equality

JavaScript has two operators for testing equality, `==` and `===`. You've probably been using `==` in your JavaScript programming, but consider using `===` instead (at least sometimes). Let's take a closer look at the difference between these two operators, and why you might use one over the other.

We'll begin by looking at an example of these two operators in action:

### INTERACTIVE SESSION:

```
> null == undefined
true
> null === undefined
false
```

The equality operator, `==`, attempts to do implied typecasting *before* it compares two values. So, in the first expression above, JavaScript sees that you're trying to compare values of two different types, and so, tries to convert one type to the other in order to do the comparison. In this case, JavaScript could either convert **undefined** to **null**, or **null** to **undefined** (JavaScript can do it either way), and then the two values are equal.

However, the strict equality operator, `===`, does *not* do implied typecasting. Instead, it compares the two values as they are. If the types of the two operands are different, then the result is false immediately. Let's see what happens when we use **strict equality** on our previous comparison of 88 with "88":

### INTERACTIVE SESSION:

```
> var myNum = 88;
undefined
> var myString = "88";
undefined
> if (myNum === myString) {
  console.log("My number is equal to my string!");
} else {
  console.log("A number shouldn't really be equal to a string!");
}
A number shouldn't really be equal to a string!
< undefined
```

Let's try **strict equality** on a falsey value, like 0:

### INTERACTIVE SESSION:

```
> var zero = 0;
undefined
> if (zero == false) {
  console.log("zero is a falsey value!");
}
zero is a falsey value!
< undefined
> if (zero === false) {
  console.log("zero is a falsey value!");
} else {
  console.log("Now we don't convert zero to false");
}
Now we don't convert zero to false
< undefined
```

So, strict equality prevents conversion of a falsey value, like 0, to false.

Just like the `==` equality operator has a negative version, `!=` (meaning *not equal to*), the strict equality operator

also has a negative version, `!==`. The only difference between `!=` and `!==` is that `!=` attempts to typecast its operands to the same type, while `!==` does not.

Do some experimenting with the four operators: `==`, `!=`, `===` and `!==`. You might be surprised by what you find.

Some programmers *always* use strict equality (and strict inequality) rather than equality (and inequality). However, sometimes you might want to take advantage of JavaScript's ability to do typecasting. If you do, be cautious and make sure you know exactly how that typecasting is going to work on the types of values you expect.

You can get all the gory details of the algorithms used by the equality operator and the strict equality operator (also known as the *identity operator*) in the [ECMAScript specification](#).

## Objects and Truthy-ness

So far, we've been looking at the truthy-ness and falsey-ness of primitive values like numbers, strings, null and undefined. What about objects?

### INTERACTIVE SESSION:

```
> var o = { name: "object" };
undefined
> o
Object {name: "object"}
> if (o) {
  console.log("This object is truthy!");
}
This object is truthy!
< undefined
```

So it looks like objects are truthy. What about empty objects? (Remember that empty strings are falsey, so you might expect that empty objects are also falsey.)

### INTERACTIVE SESSION:

```
> var p = {};
undefined
> if (p) {
  console.log("This object is truthy!");
} else {
  console.log("This object is falsey!");
}
This object is truthy!
< undefined
```

Interesting. Even a completely empty object, like `p`, is still truthy. But...

### INTERACTIVE SESSION:

```
> p == true
false
> p == false
false
```

Even though `p` might be truthy, it's not equal to `true` (or `false`) using the equality operator, so no type conversion is happening here.

## Objects and Equality

Let's do a few more tests in the console and compare our empty object, `p`, to some other values:

## INTERACTIVE SESSION:

```
> var p = {};  
undefined  
> p == 0  
false  
> p == null  
false  
> p == undefined  
false  
> p == "{}"  
false  
> p == {}  
false
```

In this example, we use the equality operator, `==`, because we want JavaScript to try to typecast the values for comparison. As you can see, all of the results are false, which means that even if JavaScript is able to typecast, the comparison is still false.

Most of these results are probably expected, but that last one sure isn't! We know that `p` is an empty object, `{ }`, so why isn't `p` equal to another empty object? Aren't they the same thing?

Go ahead and create a file with two objects so we can experiment:

## CODE TO TYPE:

```
<!doctype html>  
<html>  
<head>  
  <title> Comparing objects </title>  
  <meta charset="utf-8">  
  <script>  
    var book1 = {  
      title: "Harry Potter",  
      author: "JK Rowling",  
      published: 1999,  
      hasMovie: true  
    };  
    var book2 = {  
      title: "Harry Potter",  
      author: "JK Rowling",  
      published: 1999,  
      hasMovie: true  
    };  
  
    if (book1 == book2) {  
      console.log("The two books are the same");  
    } else {  
      console.log("The two books are different");  
    }  
  </script>  
</head>  
<body>  
</body>  
</html>
```

Here, we create two book objects using object literals, and then test to see if the books are equal.



Save this in your `/AdvJS` folder as `objectsTest.html`, and [Preview](#) . In the console, you see the message, "The two books are different."

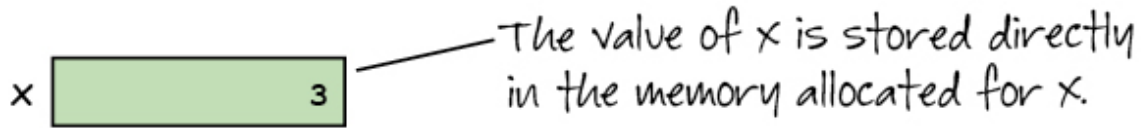
The two books, `book1` and `book2`, are exactly the same: they have the same properties. All the property names are the same, the property values are the same, and the number of properties is the same. So why aren't they equal? (Note that we're using the equality operator here to test equality; we know the types of the

two objects are the same, so we don't have to worry about any typecasting happening).

The two objects are not equal because of an important difference in how primitive values are stored and how objects are stored in the computer's memory. When you create a primitive value, let's say:

```
OBSERVE:
var x = 3;
```

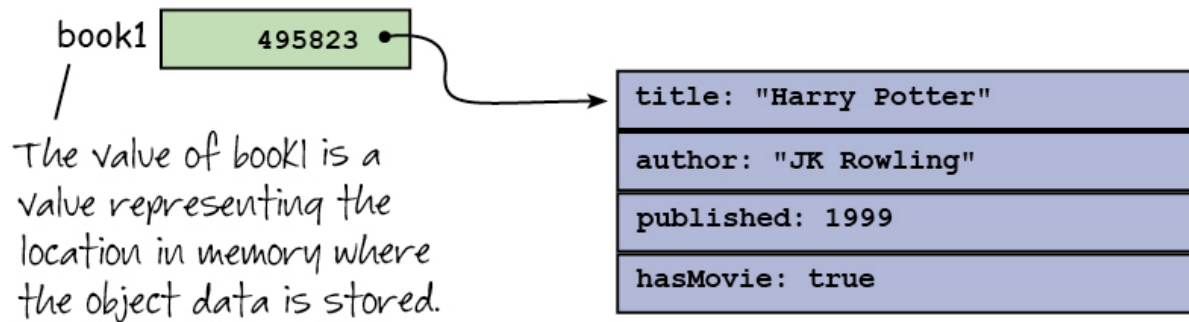
the computer allocates a bit of memory, gives it the name "x", and saves the value 3 in that bit of memory:



Now, compare that to what happens when you create an object, let's say **book1** from the example above:

```
OBSERVE:
var book1 = {
  title: "Harry Potter",
  author: "JK Rowling",
  published: 1999,
  hasMovie: true
};
```

In this case, the computer names and allocates some memory for each of the properties in the object, and then allocates a separate bit of memory for the variable name, and in that memory stores a value that points to the place in memory where the object is actually stored. This is called an **object reference**. So the variable **book1** doesn't contain the object itself; it actually contains a *reference* to the object. Like this:

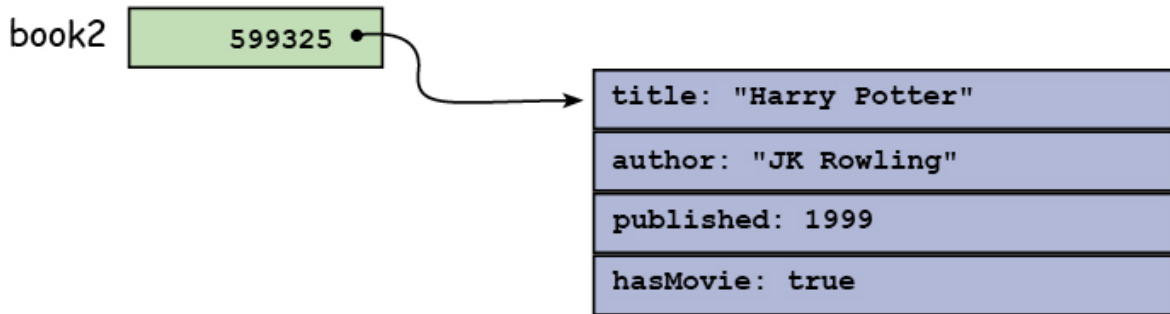


In this case, the object value (that is, all the properties in the object, plus a few other things about the object) is stored at memory location 495823 (I just made that up for this example, but you get the idea), and the variable **book1** contains that location (in the same way that **x** contains 3).

Now let's see what happens when we create the second book object, **book2**:

```
OBSERVE:
var book2 = {
  title: "Harry Potter",
  author: "JK Rowling",
  published: 1999,
  hasMovie: true
};
```

Even though the properties are exactly the same, a completely separate book object is created and stored in a completely different part of memory:

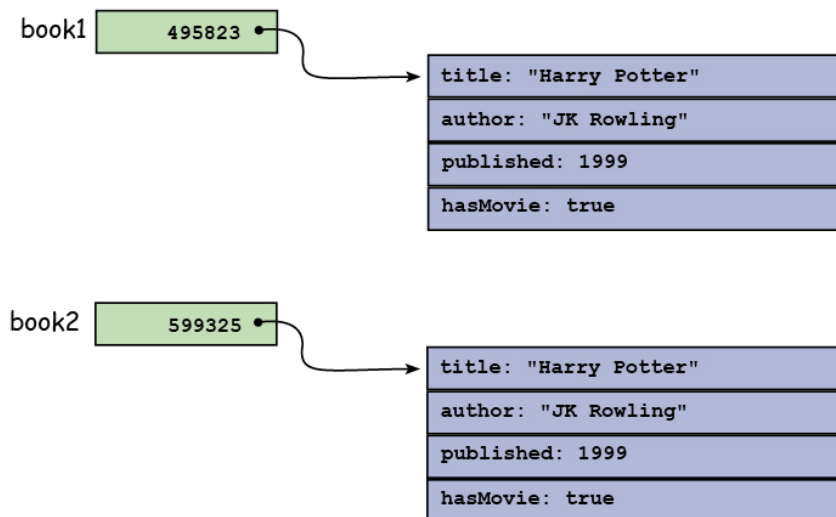


The variable **book2** contains the memory location of this second object, and the memory location is *different* from the memory location for **book1**.

So when we compare **book1** and **book2**:

```
OBSERVE:
if (book1 == book2) {
  console.log("The two books are the same");
} else {
  console.log("The two books are different");
}
```

the values that are compared are the memory locations of the two objects. They are *not* equal, so we see the message "The two books are different."

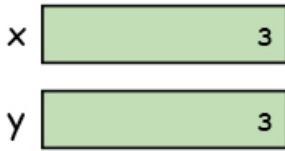


When you compare *book1* and *book2*, again JavaScript looks at the value in memory for *book1* and compares it to the value in memory for *book2*. In this case, they are different, because the data for each object is stored in a different place in memory!

Compare this to what happens when we compare primitive values:

```
INTERACTIVE SESSION:
> var x = 3;
undefined
> var y = 3;
undefined
> if (x == y) { console.log("x and y are the same"); }
x and y are the same
< undefined
```







When you compare  $x$  and  $y$ , JavaScript looks at the value in the memory for  $x$  and compares it to the value in the memory for  $y$  and if they are the same value, then  $x$  and  $y$  are the same.

In the example above where we compared two book objects, we used **literal** objects for the books. Do you think the the result would be the same when if we used an object constructor? Let's see:

```
CODE TO TYPE:
<!doctype html>
<html>
<head>
  <title> Comparing objects, take two </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;
    }
    var book1 = new Book("Harry Potter", "JK Rowling", 1999, true);
    var book2 = new Book("Harry Potter", "JK Rowling", 1999, true);
    if (book1 == book2) {
      console.log("book1 is equal to book2");
    } else {
      console.log("book1 is NOT equal to book2");
    }
  </script>
</head>
<body>
</body>
</html>
```



Now we use a constructor function, **Book()**, to create two books, and then test to see if they are equal.  Save this in your **/AdvJS** folder as **objectsTest2.html**, and **Preview** . In the console, you see the message, "book1 is NOT equal to book2."

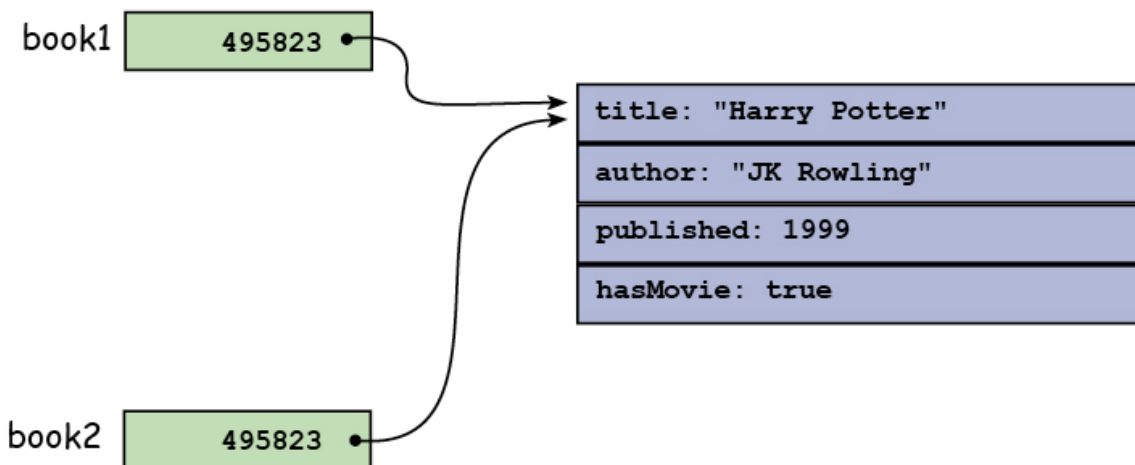
The result is the same because we are creating two completely different book objects, even though they use the same constructor and have the same properties.

Okay, so knowing what you know about how objects are stored in memory, what do you think happens when we change the program like this?:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Comparing objects, take two </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;
    }
    var book1 = new Book("Harry Potter", "JK Rowling", 1999, true);
var book2 = new Book("Harry Potter", "JK Rowling", 1999, true);
    var book2 = book1;
    if (book1 == book2) {
      console.log("book1 is equal to book2");
    } else {
      console.log("book1 is NOT equal to book2");
    }
  </script>
</head>
<body>
</body>
</html>
```

 and . In the console, you see the message "book1 is equal to book2." Why? Because when we assign the value of book1 to book2 (var book2 = book1), we store the *memory location* of the data in **book1** into **book2**, like this:




So now, when we compare the values of book1 and book2, they are the same: they are both values that point to the same memory location.

Let's update the program one more time:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Comparing objects, take two </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;
    }
    var book1 = new Book("Harry Potter", "JK Rowling", 1999, true);
    var book2 = book1;
    if (book1 == book2) {
      console.log("book1 is equal to book2");
    } else {
      console.log("book1 is NOT equal to book2");
    }
    book1.star = "Harry";
    console.log(book1);
    console.log(book2);
  </script>
</head>
<body>
</body>
</html>
```



and . Take a look at the two book objects that we display in the console. Notice anything interesting?

## OBSERVE:

```
Book {title: "Harry Potter", author: "JK Rowling", published: 1999, movie: true,
  star: "Harry"}
Book {title: "Harry Potter", author: "JK Rowling", published: 1999, movie: true,
  star: "Harry"}
```

In the code we added a new property, **star** to **book1**, and set its value to "Harry." Yet when you look at the two objects in the console, you can see that *both* **book1** and **book2** now have the property **star**, with the value "**Harry**". How did this happen!?

Well, remember that **book2** points to the same location in memory that **book1** does. So if we change the data in **book1**, we're also changing the data in **book2** because they are the *same* object.

What do you think would happen if we changed the title of **book2**? Try changing the title of **book2** to "Harry Potter and the Sorcerer's Stone." What do you see when you display **book1** and **book2**?

Now, you might be asking, "If I can't compare two different objects using the equality operator to see if they are the same (that is, that they have the same properties and values), how do I know if two objects are the same?"

The answer is that you have to look at each property of an object separately. This isn't too hard to do if the properties in an object are all primitive values (numbers, strings, booleans). However, if your objects have nested objects and/or methods, then it gets a bit trickier because then the solution depends on what you mean by "equality" in the case of two objects. What do you think it means for one object to be "equal" to another? A good topic for you to think about.

Various JavaScript libraries have tackled this question by implementing functions that check equality of objects. Be cautious though because different libraries may have different ideas about what equality of objects means. Make sure the library function works as you expect. For example, you can use the [Underscore.js](#) library's **isEqual()** function to test the equality of objects.

We covered a lot of ground in this lesson, including truthy and falsey values, implied typecasting and what can happen when you compare two values, two different kinds of equality operators, and the difference in comparing primitive values and object

values. Whew! That's lots of detail, some of which you may not have encountered before, but that you'll need to know as you get into more advanced JavaScript programming.

Take a break to rest your brain, and then tackle the quizzes and projects to help it all sink in before you move on to the next lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Constructing Objects

---

## Lesson Objectives

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

- construct objects with your own constructors, and **new**.
- construct object literals.
- construct empty objects and add new properties.
- compare how a function works as a function, and as a constructor.
- initialize an object's property values in a constructor.
- use the conditional operator.
- explore the value of **this** when an object is created.
- construct arrays in two ways.

---

Just about everything in JavaScript is an object, so understanding objects is key to understanding and programming JavaScript. In this lesson, we'll delve into how we create objects in JavaScript.

## Constructing JavaScript Objects

When you construct an object in JavaScript, you are creating a dynamic collection of property names and values. You've already seen a couple of different ways to create objects, using a constructor function (like the **Book()** function we used in the previous lesson), and using object literals.

Let's take a closer look at three ways you can construct objects and how they are similar to and different from each other.

### Constructing an Object with a Constructor Function


The first way to construct objects that we'll check out uses the a constructor function. In the previous lesson, we used a constructor function, **Book()**, to create book objects, passing in arguments for title, author, the date the book was published, and whether it had been made into a movie. We'll use that same object here, except we'll add a new method, **display()**, to the object:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Constructing objects </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;

      this.display = function() {
        console.log(this);
      };
    }
    var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur Conan Doyle", 1901, true);
    book1.display();
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **objectConstr.html**, and **Preview** . Open the console (and reload the page if you need to), and you see that the constructor function created a book object:

```

▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle",
  published: 1901, hasMovie: true, display: function} ⓘ
  ▶ author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  ▶ __proto__: Book

```

(This screenshot is in the Chrome console, but it should look similar in IE, Firefox, and Safari).

A constructor function is just like any other function, but we *call* a constructor function differently from other functions: we use the word **new**.

```

OBSERVE:
var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur Conan Doyle",
1901, true);

```

The word **new** makes all the difference. The **new** keyword indicates that we are using a function to construct an object, rather than just execute code (although a constructor function can do that too). Within the constructor function, we refer to the object that is being created as **this**:

```

OBSERVE:
function Book(title, author, published, hasMovie) {
  this.title = title;
  this.author = author;
  this.published = published;
  this.hasMovie = hasMovie;

  this.display = function() {
    console.log(this);
  };
}

```

When you call a function with **new**, inside that function, a new, empty object is created and the **this** keyword is set to that object. (Inside the function, **this** acts just like a local variable, except that you can't set its value yourself; that's done for you automatically). Then you use **this** to set the values of any properties you want in that object. At the end of the function, you don't have to explicitly return the object you're creating; JavaScript does that for you automatically because you used the **new** keyword when you called the function. The object that is returned is **this**: the new object that was created when you called the function, that has the properties you set in the function.

Of course any function *could* return an object if you wanted it to:

```

INTERACTIVE SESSION:
> function makeObj() { return { x: 1 }; }
undefined
> var myObj = makeObj();
undefined
> myObj
Object {x: 1}

```

However, in this example, **makeObj()** is *not* a constructor function because we didn't call it with **new**. Instead we called the function normally, and inside the function, created an object literal on the fly, and returned it to the caller of the function **makeObj()**. These two ways of creating functions might seem similar, but there are a

few key differences. The value of **this** inside **makeObj()** is *not* set to the object that's being created, and if you don't explicitly return an object from **makeObj()**, the default return value is **undefined**.

These differences in how functions behave, depending on whether you call them with **new** or not, is one reason why we always (as a convention) use an uppercase letter to begin the name of a constructor function (like **Book()**), but use a lowercase letter to begin the name of a regular function (like **makeObj()**). That way, you can tell at a glance at your code whether a function is designed to be a constructor function.

There's one other thing that happens when you create an object by calling a constructor function with **new** that doesn't happen when you create objects in other ways. Look back at the object we created by calling **makeObj()**:

```
> function makeObj() { return { x: 1 }; }
undefined
> var myObj = makeObj();
undefined
> myObj
Object {x: 1}
```

Now compare that to what you saw in the console earlier for the **book1** object (if you still have the page loaded, you can just type **book1** in the console to see it again, but make sure you do this in either the Chrome or Safari console specifically):

```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle",
  published: 1901, hasMovie: true, display: function} ⓘ
  ▶ author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
```

When you display **myObj** in the console, you see the word **Object** next to the object:

OBSERVE:

```
Object {x: 1}
```

But you see the word **"Book"** next to the **book1** object:

OBSERVE:

```
Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle",
  published: 1901, hasMovie: true, display: function}
```

In the **Book** example, we created the **book1** object with the **Book()** constructor function. When you create an object with a constructor function, JavaScript keeps track of that function in a property called **constructor**. **constructor** is a property of the object, in this case **book1**, that results from calling the constructor function, **Book()**. Now, JavaScript also sets the **constructor** property for objects *not* created with a constructor function, like the literal object we created and returned from the **makeObj()** function, but in this case, the **constructor** property is set to **Object()**.

You can access the constructor for an object:

## INTERACTIVE SESSION:

```
> myObj.constructor
function Object() { [native code] }
> book1.constructor
function Book(title, author, published, hasMovie) {
  this.title = title;
  this.author = author;
  this.published = published;
  this.hasMovie = hasMovie;

  this.display = function() {
    console.log(this);
  };
}
```

You can think of the constructor function of an object, whether it's **Book()** or **Object()**, as determining the *type* of the object. This isn't strictly true like it is in statically typed languages like Java, but it can be a handy way to describe objects. We'll return to this idea in a later lesson when we talk about the **instance of** operator.

For now, the key takeaway for you is to understand how constructing an object using a constructor function with the **new** keyword is different from other ways that we create objects.

## Constructing an Object Using a Literal

You just saw an example of creating a literal object and returning it from a function. Let's create another literal object, another book, so we can compare the result directly with the **book1** object we created using a constructor function. Modify **objectConstr.html** as shown:



## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Constructing objects </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;

      this.display = function() {
        console.log(this);
      };
    }
    var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur Conan Doyle", 1901, true);
    book1.display();

    var book2 = {
      title: "The Adventures of Sherlock Holmes",
      author: "Sir Arthur Conan Doyle",
      published: 1892,
      movie: true,
      display: function() {
        console.log(this);
      }
    };
    book2.display();

  </script>
</head>
<body>
</body>
</html>
```



and **Preview**. Open the console, and compare **book1** and **book2** (using Chrome or Safari):

```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book

▼ Object {title: "The Adventures of Sherlock Holmes", author: "Sir Arthur Conan Doyle", published: 1892, movie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: true
    published: 1892
    title: "The Adventures of Sherlock Holmes"
  }
  ▶ __proto__: Object

← undefined
```

These two objects are similar: both have the same property names, both have a **display()** method, and the types of all the property values are the same. However, if you look at the constructors for **book1** and **book2**, you'll see (just like in **myObj** earlier), that the constructor for **book1** is **Book()**, because we created it using a constructor function, but the constructor for **book2** is **Object()** because we created it using an object literal:

## INTERACTIVE SESSION:

```
> book1.constructor
function Book(title, author, published, hasMovie) {
  this.title = title;
  this.author = author;
  this.published = published;
  this.hasMovie = hasMovie;

  this.display = function() {
    console.log(this);
  };
}
> book2.constructor
function Object() { [native code] }
```

The **[native code]** means that the implementation of the **Object()** constructor is hidden because it's internal to the browser.

The property in the objects called **\_\_proto\_\_** is the last property in both the **book1** and **book2** objects:



```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
▼ Object {title: "The Adventures of Sherlock Holmes", author: "Sir Arthur Conan Doyle", published: 1892, movie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: true
    published: 1892
    title: "The Adventures of Sherlock Holmes"
  }
  ▶ __proto__: Object
< undefined
```

The value of **\_\_proto\_\_** for **book1** is **Book**, and the value of **\_\_proto\_\_** for **book2** is **Object**. You might be thinking that the **\_\_proto\_\_** property must be related to the constructor function for an object, and you'd be right (although they are *not* the same thing).

You also might notice that the **constructor** property is *not* listed in the **book1** and **book2** objects' properties. That's because this property is *inherited* from the object's **prototype**. (As you might guess, the **\_\_proto\_\_** property is also related to the prototype). We'll talk more about prototypes in a later lesson.

What about **this** in a literal object? You already know that you can use **this** in a method of an object to refer to "this object," but unlike in a constructor function, we don't (and can't) use **this** to initialize object properties. Instead, we create properties and initialize them by specifying the name/value pairs in an object, by literally typing them. (That's why it's called an object literal). The only time **this** refers to the object is when you call one of its methods. (How **this** gets set and what it gets set to is yet another topic we'll come back to in more depth later).

## Constructing an Object Using a Generic Object Constructor

Another way to create an object literal is to start with an empty object, and then add properties to it. You can create an empty, generic object in one of two ways:

### OBSERVE:

```
var obj1 = { };
var obj2 = new Object();
```

Both of these approaches to create an object do the same thing. You'll see an empty object created the first way more often (because it's a little easier to write), but it's instructive to understand the second way as well. When you create an object with **new Object()**, it's just like when you create an object with **new Book()**, except that **Object()** is a built-in constructor function that you don't have to write yourself. You don't pass any arguments to **Object()** to initialize object properties in the constructor function, instead, you add them all after the object is created. Modify **objectConstr.html** as shown:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Constructing objects </title>
  <meta charset="utf-8">
  <script>
    function Book(title, author, published, hasMovie) {
      this.title = title;
      this.author = author;
      this.published = published;
      this.hasMovie = hasMovie;

      this.display = function() {
        console.log(this);
      };
    }
    var book1 = new Book("The Hound of the Baskervilles", "Sir Arthur Conan Doyle", 1901, true);
    book1.display();

    var book2 = {
      title: "The Adventures of Sherlock Holmes",
      author: "Sir Arthur Conan Doyle",
      published: 1892,
      movie: true,
      display: function() {
        console.log(this);
      }
    };
    book2.display();

    var book3 = new Object(); // same as var book3 = { };
    book3.title = "A Study in Scarlet";
    book3.author = "Sir Arthur Conan Doyle";
    book3.published = 1887;
    book3.movie = false;
    book3.display = function() {
      console.log(this);
    };
    book3.display();

  </script>
</head>
<body>
</body>
</html>
```

We added the exact same properties that we added to **book1** and **book2**, only with some different values, because it's a different book. **book3** also has a **display()** method, just like **book1** and **book2**.



and



. In the console, compare **book3** with **book2** and **book1**.

```
▼ Book {title: "The Hound of the Baskervilles", author: "Sir Arthur Conan Doyle", published: 1901, hasMovie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    hasMovie: true
    published: 1901
    title: "The Hound of the Baskervilles"
  }
  ▶ __proto__: Book
▼ Object {title: "The Adventures of Sherlock Holmes", author: "Sir Arthur Conan Doyle", published: 1892, movie: true, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: true
    published: 1892
    title: "The Adventures of Sherlock Holmes"
  }
  ▶ __proto__: Object
▼ Object {title: "A Study in Scarlet", author: "Sir Arthur Conan Doyle", published: 1887, movie: false, display: function} ⓘ
  author: "Sir Arthur Conan Doyle"
  ▶ display: function () {
    movie: false
    published: 1887
    title: "A Study in Scarlet"
  }
  ▶ __proto__: Object
```

**book3** looks similar to **book2**, because **book3** is also an object literal; it was just created in a slightly different way. Notice that the constructor for **book3** is also **Object()**, which you can test in the browser console:

```
INTERACTIVE SESSION:
> book3.constructor
function Object() { [native code] }
```

## So, What's the Best Way to Make an Object?

You've seen three different ways to construct an object (you'll see a fourth in a later lesson), but which is the *best* way?

That depends on the situation. If all you need is a quick, one-off object, then creating an object literal like we did with **book2** or **book3** is probably good enough. However, if you know that you're going to need multiple book objects, writing a constructor function like **Book()**, that you can use to make many book objects is a better choice. You'll also want to consider whether the objects you're creating are, say, **Books** or **Magazines**. As you've seen, objects created with a constructor function have that extra information about how they were created, which can be useful. We'll see an example of that in a later lesson, when we talk about prototypes.

## Initializing Values in Constructors

Let's go back to constructor functions now and look at ways you can initialize the properties of the object you're constructing with the function. We'll use a different example, so open a new file:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Initializing objects </title>
  <meta charset="utf-8">
  <script>
    function Point(x, y) {
      if (x == undefined || x == null) {
        this.x = 50;
      } else {
        this.x = x;
      }

      if (y == undefined || y == null) {
        this.y = 50;
      } else {
        this.y = y;
      }

      // Make a toString() method we can use to display the point
      this.toString = function() {
        return "[" + this.x + ", " + this.y + "]";
      }
    }

    // Can have code in constructors too
    var p = new Point();
    console.log("My point is: " + p.toString());
  </script>
</head>
<body>
</body>
</html>
```



Save it in your **/AdvJS** folder as **point.html**, and **Preview**. Open the console (and reload the page if you need to); the Point object, **p**, is displayed like this:

#### OBSERVE:

```
My point is: [50, 50]
```

In this code, we've got a **Point()** constructor function to create Point objects. Let's discuss the code:

**OBSERVE:**

```
function Point(x, y) {
  if (x == undefined || x == null) {
    this.x = 50;
  } else {
    this.x = x;
  }

  if (y == undefined || y == null) {
    this.y = 50;
  } else {
    this.y = y;
  }

  // toString(): display the point
  this.toString = function() {
    return "[" + this.x + ", " + this.y + "]";
  }
}

var p = new Point();
console.log("My point is: " + p.toString());
```

We use a **constructor function, Point()** to create a Point object, **p**. So, if you look at the constructor property of the object **p**, you'll see the **Point()** function.

**Point()** actually expects two arguments, **x and y**, which are the coordinates of the point, but we call **Point()** with no arguments. It turns out that JavaScript is totally okay with this, but unless we do something further, the **x** and **y** coordinates of the point we're trying to create will be undefined. So, we write code to test to see whether the **x** and **y** values are passed in. If they are, we initialize **this.x** and **this.y** with the values **x** and **y** respectively. If we don't pass in any values, we use the default value, **50** for both **this.x** and **this.y**.

Now, you might be tempted to take a shortcut and rewrite **if (x == undefined || x == null) { ... }** as **if (!x) { ... }** (based on what you learned in the previous lesson about truthy and falsey values), but be careful! We might want a point at 0, 0, and 0 is falsey, so that shortcut won't work for us in this case. We can, however, shorten the initialization a bit by making use of the **conditional operator**. Modify **point.html** as shown:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Initializing objects </title>
  <meta charset="utf-8">
  <script>
    function Point(x, y) {
      this.x = (!x && x != 0) ? 50 : x;
      this.y = (!y && y != 0) ? 50 : y;

      if (x == undefined || x == null) {
        this.x = 50;
      } else {
        this.x = x;
      }

      if (y == undefined || y == null) {
        this.y = 50;
      } else {
        this.y = y;
      }

      // Make a toString() method we can use to display the point
      this.toString = function() {
        return "[" + this.x + ", " + this.y + "]";
      }
    }

    // Can have code in constructors too
    var p = new Point();
    console.log("My point is: " + p.toString());
  </script>
</head>
<body>
</body>
</html>
```



and **Preview**. You see the same result as before.

Now, some programmers avoid the conditional operator (also written `?:` for short) like the plague, because it's harder to read, and you can always write the same code using an easier-to-read `if/else` statement. If you're in this camp, feel free to use `if/else` statements instead. Still, you need to know how to read statements that use the conditional operator too; they are used fairly often to initialize objects.

So, you read this:

## OBSERVE:

```
this.x = (!x && x != 0) ? 50 : x;
```

like this: "If not **x** **AND** **x** is not equal to 0, **THEN** set **this.x** to 50 **ELSE** set **this.x** to **x**."

In other words, you read `?` as THEN and `:` as ELSE.

This statement checks to see if `!x` is true, which it will be if the parameter `x` is null, undefined, or 0. Then, to handle the 0 case, we check to make sure `x != 0`. If `x` is 0, this returns false, so the whole conditional is false, and we set `this.x` equal to `x`, which is 0 in this case. If `x` is undefined or null, we set `this.x` to 50. If `x` is a non-zero number we set `this.x` to `x`.

Don't get the *parameter* `x` mixed up with the *property* `this.x`. They are two different variables! Remember that we're passing a value into the constructor to initialize the property `this.x`. The value we pass in gets assigned to the parameter `x`.

Each `Point` object also has a method, `toString()`, that creates a string representation of the point for display. In `toString()` we use `this.x` and `this.y` to create the string representing the `Point`. Make sure to use the

object's properties, and *not* the parameters in the method. If you forget, and use **x** and **y** instead of **this.x** and **this.y**, what could happen? Well, if **Point()** is called with no arguments, then the parameters **x** and **y** will be undefined. While the Point's **x** and **y** properties are set correctly to 50 each, you'll see [ **undefined**, **undefined** ] when you call the **toString()** method.

## **this**

We've talked a bit about what happens to **this** when you're constructing objects. To make sure you've got a handle on **this** when you're working with constructor functions, regular functions, objects, and methods, let's take a look at another example. Create this new file:



**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> What happens to this </title>
  <meta charset="utf-8">
  <script>
    //
    // Rectangle constructor that makes rectangle objects
    //
    function Rectangle(width, height) {
      console.log("This in Rectangle is: ");
      console.log(this);

      this.width = width || 0;
      this.height = height || 0;
      this.getArea = function() {
        console.log("This in Rectangle's getArea is: ");
        console.log(this);
        return this.width * this.height;
      };
    }

    var rect1 = new Rectangle(5, 10);
    console.log("Area of rectangle 1: " + rect1.getArea());

    //
    // A function that makes rectangle objects
    //
    function makeRectangle(width, height) {
      console.log("This in makeRectangle is: ");
      console.log(this);

      return {
        width: width || 0,
        height: height || 0,
        getArea: function() {
          console.log("This in makeRectangle's getArea is: ");
          console.log(this);
          return this.width * this.height;
        }
      };
    }

    var rect2 = makeRectangle(5, 10);
    console.log("Area of rectangle 2: " + rect2.getArea());

    // getArea function
    function getArea(r) {
      console.log("This in getArea is: ");
      console.log(this);
      return (r.width * r.height);
    }
    console.log("Area from getArea(rect1): " + getArea(rect1));

  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **rectangle.html** and **Preview** . In the console, you see lots of output:

```

This in Rectangle is:
Rectangle {}
This in a Rectangle's getArea is:
▶ Rectangle {width: 5, height: 10, getArea: function}
Area of rectangle 1: 50
This in makeRectangle is:
▶ Window {top: Window, window: Window, location: Location, external: Object, chrome: Object...}
This in a rectangle's getArea is:
▶ Object {width: 5, height: 10, getArea: function}
Area of rectangle 2: 50
This in getArea is:
▶ Window {top: Window, window: Window, location: Location, external: Object, chrome: Object...}
Area from getArea(rect1): 50

```

There's quite a bit going on here, so we'll step through it one piece at a time:

OBSERVE:

```

//
// Rectangle constructor that makes rectangle objects
//
function Rectangle(width, height) {
  console.log("This in Rectangle is: ");
  console.log(this);

  this.width = width || 0;
  this.height = height || 0;
  this.getArea = function() {
    console.log("This in a Rectangle's getArea is: ");
    console.log(this);
    return this.width * this.height;
  };
}
var rect1 = new Rectangle(5, 10);
console.log("Area of rectangle 1: " + rect1.getArea());

```

In this code, we've got a **Rectangle()** constructor function that we can use to make rectangles with **width** and **height** properties, and a method, **getArea()** that returns the area of the rectangle. We've added calls to **console.log()** in two different places within the constructor function to inspect the value of **this**.

When we call **new Rectangle(5, 10)** to create a rectangle object, **rect1**, the **first two lines of code in the function** display the value of **this**. The result is an empty **Rectangle** object:

OBSERVE:

```

This in Rectangle is:
Rectangle {}

```

When we call a construction function with **new**, the first thing that happens is a new, empty object is created. This is the **Rectangle {}** object we see here in the console. Its constructor is **Rectangle**, and it doesn't have any properties (yet).

The rest of the constructor function assigns values to the **width**, **height** and **getArea()** properties, so that when the object is returned at the end of the function, all of its properties have been created and given values.

Next, we **call the getArea() method of the rectangle object we just created**. The **first two lines of the getArea() method** display the value of **this** in the console. We see that **this** is a **Rectangle** object, and that now it has the properties we created in the constructor:

OBSERVE:

```
This in Rectangle is:
Rectangle {}
This in Rectangle's getArea is:
Rectangle {width: 5, height: 10, getArea: function}
Area of rectangle 1: 50
```

Finally, we display the result of the call to `getArea()`, which is 50.

So in both the body of the constructor function and the method, `this` refers to "this object," that is, the `Rectangle` object created by the constructor. The first use of `this` is the object when it's created and modified at object creation time (when we call the constructor function). The second use of `this` is the object when it's accessed after we call the object's method, `getArea()`. In this case, the value of `this` is assigned automatically because you are calling a method of an object.

Now let's compare that to what happens when we call `makeRectangle()` to make a rectangle object.

OBSERVE:

```
//
// A function that makes rectangle objects
//
function makeRectangle(width, height) {
  console.log("This in makeRectangle is: ");
  console.log(this);

  return {
    width: width || 0,
    height: height || 0,
    getArea: function() {
      console.log("This in makeRectangle's getArea is: ");
      console.log(this);
      return this.width * this.height;
    }
  };
}

var rect2 = makeRectangle(5, 10);
console.log("Area of rectangle 2: " + rect2.getArea());
```

`makeRectangle()` isn't a constructor function, it's just a regular function, so we don't call it with `new`; we just call it the regular way. The **first two lines of code in `makeRectangle()`** display the value of `this`. You can see that `this` is the global, `Window` object:

OBSERVE:

```
This in makeRectangle is:
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
This in a makeRectangle's getArea is:
Object {width: 5, height: 10, getArea: function}
Area of rectangle 2: 50
```

There is no object being created automatically by `makeRectangle()`. While we are creating an object in this function, that object is not the value of `this`. We create that object in the next statement (by returning an object literal).

However, when we **call the `getArea()` method** of the object returned by `makeObject()`, `rect2`, you can see that the value of `this` in the `getArea()` method is indeed an object:

OBSERVE:

```
This in makeRectangle is:  
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}  
This in makeRectangle's getArea is:  
Object {width: 5, height: 10, getArea: function}  
Area of rectangle 2: 50
```

The object that we see is **rect2**, "this object," that is, the object whose method we called. Again, notice that the object's constructor is **Object()** (compare to the constructor for **rect1** above).

Finally, we display the area for **rect2**, which is 50.

We've also included a function **getArea()** that takes a rectangle object and returns the area:

OBSERVE:

```
// getArea function  
function getArea(r) {  
  console.log("This in getArea is: ");  
  console.log(this);  
  return (r.width * r.height);  
}  
console.log("Area from getArea(rect1): " + getArea(rect1));
```

The value of **this** in the **getArea()** function displays as:

OBSERVE:

```
This in getArea is:  
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}  
Area from getArea(rect1): 50
```

Once again, **this** is the **global Window object**. Just like **makeRectangle()**, **getArea()** is just a regular old function that happens to take an object and compute something with it. There is no "this object" for this function, so the value of **this** gets set to the Window object automatically.

Keeping track of the value of **this** can be tricky in JavaScript, but it's important. You'll need to understand how the value of **this** is set, and what it's set to in all situations. We'll also revisit **this** in later lessons.

## Constructing Array Objects

Before we leave this lesson, let's talk about constructing Array objects. Arrays are objects, although you should think of them as a special kind of object with features that the objects we've been creating so far don't have, like an index, and ordering imposed on the items in the object.

There are two ways to create an Array object. Type these commands in the console:

INTERACTIVE SESSION:

```
> var a1 = new Array();  
undefined  
> a1[0] = 1;  
1  
> a1[1] = 2;  
2  
> a1[2] = 3;  
3  
> a1  
[1, 2, 3]
```

Here we created an empty array, **a1**, using the **Array()** constructor function, calling it with **new** like we would

any other constructor function. Then we add array items one at a time to the 0, 1, and 2 indices in the array. This is analogous to using **new Object()** and adding object properties one at a time, like we did with **book3** earlier in the lesson.

You can use bracket notation to access an object's properties, like this:

OBSERVE:

```
var theTitle = book1["title"];
```

the bracket notation is used to access the items in an array, except we use an index instead of a property name.

The second way to create an Array is to use the array literal notation:

INTERACTIVE SESSION:

```
> var a2 = [1, 2, 3];
undefined
> a2
[1, 2, 3]
```

This does exactly the same thing as the previous example; it creates a new array with values at the 0, 1, and 2 indices, but it's a lot shorter to write! In practice, you'll rarely use the **Array()** constructor to create an array. Instead you'll use the more concise array literal notation. One exception is when you need to create an empty array with a predefined number of indices:

INTERACTIVE SESSION:

```
> var a3 = new Array(100);
undefined
> a3
[undefined x 100]
```

This creates an array with length 100, with all the items at every index set to **undefined**, and the Chrome console uses the shorthand "[undefined x 100]" to display the value of this array. (Other browsers don't use this shorthand, so you'll see different results in different browsers when you ask for the value of **a3**).

Just like any other object, arrays can have named properties, and in fact, come with a named property, **length**, that you'll use to get the length of your array:

INTERACTIVE SESSION:

```
> a1.length
3
> a2.length
3
> a3.length
100
```

Just like other objects, you can use the **constructor** property to inspect the constructor function for the array:

## INTERACTIVE SESSION:

```
> a1.constructor
function Array() { [native code] }
> a2.constructor
function Array() { [native code] }
> a3.constructor
function Array() { [native code] }
```

In each case, the constructor for the array is **Array()**. This is analogous to **Object()** being the constructor for objects created with literal notation or with **new Object()**.

In this lesson, you learned about constructing objects, how constructor functions work, the difference between objects created with a constructor function and those created using literal notation, and what happens to **this** when you construct and use an object.

In the next lesson, we'll explore more object-related goodies: prototypes and inheritance. Before you dive in though, do the quizzes and projects, and then take a well-earned break.

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Prototypes and Inheritance

---

## Lesson Objectives

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

- use `instanceof` to check the constructor, or "type," of a specific object.
- examine the prototype property of a constructor function.
- add methods to an object's prototype to share them among objects.
- explore what happens to `this` in an object's prototype.
- examine an object's prototype chain.
- use prototypical inheritance to access properties from higher up the prototype chain.
- determine if a property is defined in an object or an object's prototype with `hasOwnProperty()`.
- examine the prototype of an object in the console using `__proto__`.

---

JavaScript is an "object-oriented language" in two ways: first it's object-oriented in that just about everything in the language is an object (except for a few primitives). Second, it's object-oriented in the sense that objects can inherit properties from other objects and, thus, share code with them. However, if you have had any experience with a language like Java or C++ or C#, be prepared to think differently in this lesson, because JavaScript objects inherit properties differently than those languages do.

## Object-Oriented Programming in JavaScript

The key to understanding JavaScript objects work, and how they inherit properties, is to understand **object prototypes**. Before we jump into prototypes though, let's review how objects are created with constructor functions, and also introduce a new operator, **instanceof**.

### instanceof

Let's begin our in-depth study of objects by creating an example. We'll make this example a bit more interesting and display a representation of the objects in the web page (the point is to understand how objects work though, so don't get too caught up in the cool web page part of this example). Create a new HTML file as shown, and then go through it, step by step:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Shapes with Prototypes and Inheritance </title>
  <meta charset="utf-8">
  <style>
    html, body, div#container {
      width: 100%;
      height: 100%;
      margin: 0px;
      padding: 0px;
    }
    div#container {
      position: relative;
    }
    .shape {
      position: absolute;
      text-align: center;
    }
    .shape span {
      position: relative;
      top: 44%;
    }
    .square {
      background-color: lightblue;
    }
    .circle {
      background-color: goldenrod;
      border-radius: 50%;
    }
  </style>
  <script>
function Circle(name, radius) {
  this.name = name;
  this.radius = radius;
  this.getCircumference = function() {
    return this.radius * Math.PI * 2;
  };
  this.getName = function() {
    return this.name;
  };
}

function Square(name, size) {
  this.name = name;
  this.size = size;
  this.getArea = function() {
    return this.size ^ 2;
  };
  this.getName = function() {
    return this.name;
  };
}

// Global variables so we can inspect them
// easily in the console! (Otherwise, we'd normally
// make them local to the window.onload function).
var circle1 = new Circle("circle1", 100);
var circle2 = new Circle("circle2", 200);
var square = new Square("my square", 150);

window.onload = function() {
  addShapeToPage(circle1);
  addShapeToPage(circle2);
  addShapeToPage(square);
};
```



```

function addShapeToPage(shape) {
  var container = document.getElementById("container");
  var div = document.createElement("div");
  var width = 0;
  var classes = "shape ";
  if (shape instanceof Circle) {
    classes += "circle";
    width = shape.radius;
  } else if (shape instanceof Square) {
    classes += "square";
    width = shape.size;
  }
  div.setAttribute("class", classes);
  div.style.left = Math.floor(Math.random() * (container.offsetWidth - 175)) +
"px";
  div.style.top = Math.floor(Math.random() * (container.offsetHeight - 175)) +
"px";
  div.style.width = width + "px";
  div.style.height = width + "px";


  var span = document.createElement("span");
  span.innerHTML = shape.getName();
  span.style.visibility = "hidden";
  div.appendChild(span);

  div.onmouseover = function() {
    // this is the div (the shape) you click on
    this.firstChild.style.visibility = "visible";
  };

  container.appendChild(div);
}
</script>
</head>
<body>
<div id="container"></div>
</body>
</html>

```



Save this in your **/AdvJS** folder as **proto.html**, and  **Preview**. You see three shapes on the page: two circles, and a square. Open the console, and verify that you can access the three global variables we created for the three shapes:

#### INTERACTIVE SESSION:

```

> circle1
Circle {name: "circle1", radius: 100, getCircumference: function, getName: function}
> circle2
Circle {name: "circle2", radius: 200, getCircumference: function, getName: function}
> square
Square {name: "my square", size: 150, getArea: function, getName: function}

```

We have much to discuss in this code, including a new operator, **instanceof**. We'll go over it one chunk at a time.

First, we have two constructor functions, **Circle()** and **Square()**:

OBSERVE:

```
function Circle(name, radius) {
  this.name = name;
  this.radius = radius;
  this.getCircumference = function() {
    return this.radius * Math.PI * 2;
  };
  this.getName = function() {
    return this.name;
  }
}

function Square(name, size) {
  this.name = name;
  this.size = size;
  this.getArea = function() {
    return this.size ^ 2;
  };
  this.getName = function() {
    return this.name;
  }
}
```

**Circle()** and **Square()** are similar to other constructor functions we've worked with in this course. The two constructors are almost identical; both have a **name** property and a **getName()** method. **Circle()** has a **radius** property and a **getCircumference()** method, while **Square()** has a **size** property and a **getArea()** method.

Next, we create three objects from the two constructors: two circles and a square:

OBSERVE:

```
var circle1 = new Circle("circle1", 100);
var circle2 = new Circle("circle2", 200);
var square = new Square("my square", 150);
```

We pass in initial values for the **name** property, and the **radius** and **size** properties for the circles and square, respectively. We can inspect these global variables in the console.

When we display the three objects in the console, we use a constructor function to make each kind of shape, we know that the circles are made from the **Circle()** constructor and the square is made from the **Square()** constructor:

OBSERVE:

```
> circle1
Circle {name: "circle1", radius: 100, getCircumference: function, getName: function}
> circle2
Circle {name: "circle2", radius: 200, getCircumference: function, getName: function}
> square
Square {name: "my square", size: 150, getArea: function, getName: function}
```

We have a short function assigned to the **window.onload** property that will run once the page is loaded into the browser and the DOM is ready. This function calls the **addShapeToPage()** function for each shape we've created.

**OBSERVE:**

```
function addShapeToPage(shape) {
    var container = document.getElementById("container");
    var div = document.createElement("div");
    var width = 0;
    var classes = "shape ";
    if (shape instanceof Circle) {
        classes += "circle";
        width = shape.radius;
    } else if (shape instanceof Square) {
        classes += "square";
        width = shape.size;
    }
    div.setAttribute("class", classes);
    div.style.left = Math.floor(Math.random() * (container.offsetWidth - 175)) +
    "px";
    div.style.top = Math.floor(Math.random() * (container.offsetHeight - 175)) +
    "px";
    div.style.width = width + "px";
    div.style.height = width + "px";

    var span = document.createElement("span");
    span.innerHTML = shape.getName();
    span.style.visibility = "hidden";
    div.appendChild(span);

    div.onmouseover = function() {
        // this is the div (the shape) you click on
        this.firstChild.style.visibility = "visible";
    };

    container.appendChild(div);
}
```

The `addShapeToPage()` function adds the shapes to the page. It has a `shape` parameter, which is either a circle or a square, and creates a `<div>` element for that shape. Of course, we want our circles to look like circles and our squares to look like squares, so we've created CSS classes to style the `<div>` elements for each kind of shape appropriately. Take a look back at the CSS and you'll see that we have a `shape` class for both kinds of shapes, a `square` class specifically for squares, and a `circle` class specifically for circles:

**OBSERVE:**

```
.shape {
    position: absolute;
    text-align: center;
}
.shape span {
    position: relative;
    top: 44%;
}
.square {
    background-color: lightblue;
}
.circle {
    background-color: goldenrod;
    border-radius: 50%;
}
```

In `addShapeToPage()`, we need to know if the `shape` that was passed in is a circle or a square. Why? Because if it's a circle, we want to add the `circle` class to the `<div>`, if it's a square, we want to add the `square` class to the `<div>`. In addition, in order to set the width of the `<div>` correctly, we'll need to access either the `radius` property for circles or the `size` property for squares.

So how do we determine if the `shape` that got passed in is a square or a circle? We use the `instanceof` operator:

#### OBSERVE:

```
var width = 0;
var classes = "shape ";
if (shape instanceof Circle) {
    classes += "circle";
    width = shape.radius;
} else if (shape instanceof Square) {
    classes += "square";
    width = shape.size;
}
```

The **instanceof** operator is a binary operator: it takes two arguments. The operand on the left is the object (you want to determine the type of that object) and the operand on the right is the name of the constructor function used to create the object. In this case, we use the **instanceof** operator to find out if the **shape** is a **Circle**. If it is, we know that the **shape** was constructed using the **Circle()** constructor function, and it's a circle object. Similarly, if it's **Square**, we know that **shape** is a square.

Once we know whether the **shape** is a circle or a square, we can set the **classes** and **width** variables, so the shapes will display correctly in the web page.

Then we set up the rest of the `<div>` to display a circle or square in the web page, and add the `<div>` to the DOM so it displays in the page. We add a **mouseover handler** to the `<div>`. About that handler:

#### OBSERVE:

```
div.onmouseover = function() {
    // this is the div (the shape) you click on
    this.firstChild.style.visibility = "visible";
};
```

In the previous lesson we talked about **this** and what **this** is set to when you are constructing an object, or calling the method of an object.

In most circumstances, when an event handler that is attached to a DOM object is called, **this** is set to the DOM object on which that event took place. So in this case, we attached a mouseover handler to the `<div>` object that represents our shape. When you mouse over that `<div>`, and the handler function is called, **this** is set to the `<div>` object, *not* the **shape** object.

This can trip you up really easily, and yet another reason it's important to keep track of what **this** is.

## Prototypes

So what are prototypes anyway?

Whenever you create an object in JavaScript, you get a second object with it, its **prototype**. The prototype is associated with the constructor of an object. Every function has a property, **prototype**, that holds a prototype object. Whenever you use that function as a constructor to create a new object, that new object gets the object in that function's **prototype** property as its prototype. So, the **Circle()** constructor function has a **prototype** property that contains a **Circle** prototype object. If you use **Circle()** to create a new object, **circle1**, **circle1**'s prototype will be **Circle.prototype** (that is, the object in the **prototype** property of the **Circle()** constructor).

You can take a look at the prototype of a constructor function by using the **prototype** property of the function, like this:

## INTERACTIVE SESSION:

```
> Circle
function Circle(name, radius) {
  this.name = name;
  this.radius = radius;
  this.getCircumference = function() {
    return this.radius * Math.PI * 2;
  };
  this.getName = function() {
    return this.name;
  }
}
> Circle.prototype
Circle {}
> Square
function Square(name, size) {
  this.name = name;
  this.size = size;
  this.getArea = function() {
    return this.size ^ 2;
  };
  this.getName = function() {
    return this.name;
  }
}
> Square.prototype
Square {}
```

When you type `Circle` at the console, you see its value is the constructor function, `Circle()`. When you type in `Circle.prototype`, you'll see its value is an *object*, `Circle {}`. The `Circle.prototype` object is an empty object; there's nothing in it. However, any object you make using the `Circle()` constructor gets this `Circle {}` object as its **prototype**.

You can find out the prototype of an object by getting the prototype of that object's constructor, like this:

## INTERACTIVE SESSION:

```
> circle1.constructor.prototype
Circle { }
```

Try getting the prototype of the `circle2` and `square` objects using the console. Do you get the result that you expect?

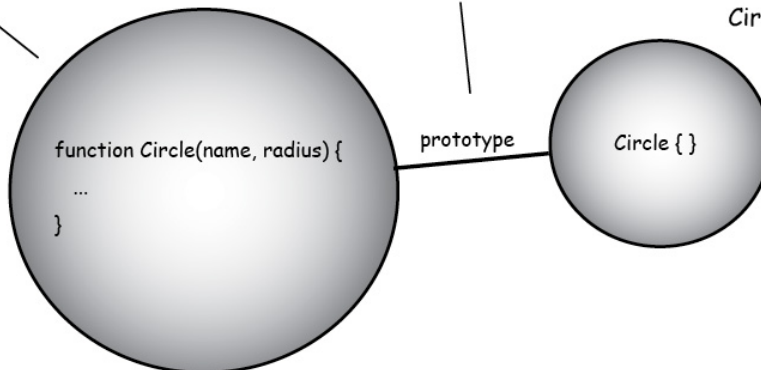
It can get a little confusing at first to keep all this straight. But just think of it like this: whenever you make an object, that object gets a prototype. A prototype is just an object! And if you want to access an object's prototype, you use that object's constructor function's **prototype** property.

So, when you write `new Circle("circle1", 100)` you get back an object, `circle1`. The `circle1` object's prototype is another object, `Circle.prototype`. Note that while we say `circle1`'s prototype is `Circle.prototype`, that doesn't mean that `circle1` has a *property* named **prototype** (it doesn't). The relationship between `circle1` and its prototype is not the same as `circle1` having a property. Now, you can *get to* the prototype of `circle1` through `circle1`'s constructor function, `Circle()`, using the constructor function's **prototype** property: `Circle.prototype`, but that's a different thing. Maybe this diagram will help make the distinction more clear:

Here's our Circle constructor function. It's a function, but it's also an object.

The Circle function (Object) has a property, `prototype`.

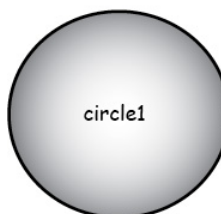
The `prototype` property's value is another object, `Circle.prototype`.



When you create an object from the Circle constructor using `new`...

```
var circle1 = new Circle("circle1", 100);
```

... you get an object whose prototype is `Circle`.



The prototype of `circle1` is `Circle.prototype`. Note that `Circle.prototype` is not a property of `circle1`! It's `circle1`'s prototype.

Now, because you are creating the object `circle2` using the same constructor function, `Circle()`, `circle2` has the same prototype as `circle1`:

#### INTERACTIVE SESSION:

```
> circle1.constructor.prototype  
Circle { }  
> circle2.constructor.prototype  
Circle { }  
> circle1.constructor.prototype === circle2.constructor.prototype  
true
```

I should point out that the `prototype` property is a property of the `Circle()` function. Yes, functions can have properties! Why? Because functions are objects! They are special objects, but they are objects just the same. We'll talk about this in more detail a little later.

## Prototypes of Literal Objects

Let's see an example that uses the prototype of a literal object:

## INTERACTIVE SESSION:

```
> var myObject = { x: 3 };
undefined
> myObject
Object {x: 3}
> myObject.constructor
function Object() { [native code] }
> myObject.constructor.prototype
Object { }
> myObject instanceof Object
true
> myObject instanceof Circle
false
```

The literal **myObject** object's constructor is **Object()**, and its prototype is **Object.prototype**. We can check to see if **myObject** is an **Object** using **instanceof**. For good measure, we check to see if **myObject** is a **Circle**, and it's not. Good!

## What is a Prototype Good For?

All this talk about prototypes, and you're probably still wondering, what good is a prototype? Why would I care if an object has a prototype?

This is where "object-oriented programming" comes in. A huge advantage of a prototype is that you can put properties that will be shared across all objects and have that prototype, in the prototype object.

When you try to access a property of an object, whether that property contains a primitive value or a method, JavaScript first looks for that property in the object; if JavaScript doesn't find the property there, it looks in the prototype. If it finds the property there, that's the value JavaScript uses. This is called the **prototype chain**. Let's take a closer look at how this works.

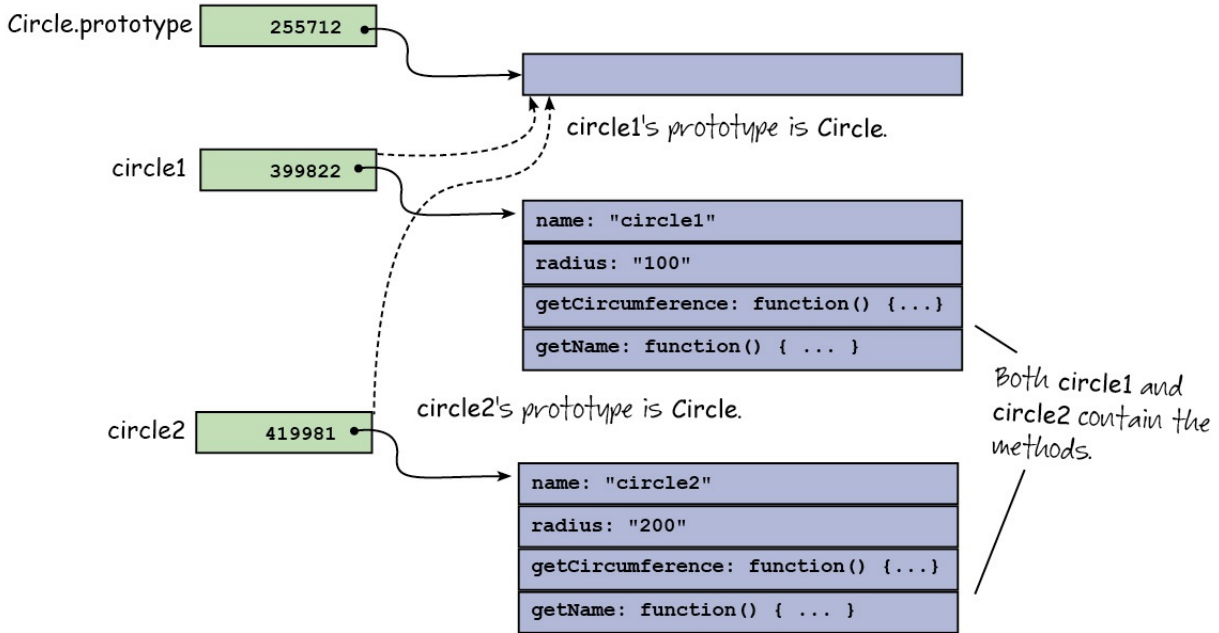
**circle1** and **circle2** each have a different name, and a different radius, but the two methods, **getCircumference()** and **getName()**, are the *same* for both objects—they don't change.

However, when you create two separate objects from the same constructor, like we did:

### OBSERVE:

```
var circle1 = new Circle("circle1", 100);
var circle2 = new Circle("circle2", 200);
```

each object gets its own copy of the **getCircumference()** and **getName()** methods:



In this example, it doesn't matter so much, but in an instance where each object has a large number of properties or methods, it can become problematic. Each object takes up memory. The more memory the object takes up, the more memory your program uses and the slower it will be.

The **Circle.prototype** of both **circle1** and **circle2** is currently an empty object:

```

INTERACTIVE SESSION:

> circle1.constructor.prototype
Circle { }

```

Similarly, the **Square.prototype** is currently an empty object.

We can move the two methods that both circles contain into the **Circle.prototype** object, and we can move the methods that all squares get into the **Square.prototype** object. Modify **proto.html** as shown:



## CODE TO TYPE:

```
function Circle(name, radius) {
    this.name = name;
    this.radius = radius;
    this.getCircumference = function() {
        return this.radius * Math.PI * 2;
    };
    this.getName = function() {
        return this.name;
    };
}
Circle.prototype.getCircumference = function() {
    return this.radius * Math.PI * 2;
};
Circle.prototype.getName = function() {
    return this.name;
};

function Square(name, size) {
    this.name = name;
    this.size = size;
    this.getArea = function() {
        return this.size ^ 2;
    };
    this.getName = function() {
        return this.name;
    };
}
Square.prototype.getArea = function() {
    return this.size * this.size;
};
Square.prototype.getName = function() {
    return this.name;
};

// Global variables so we can inspect them
// easily in the console! (Otherwise, we'd normally
// make them local to the window.onload function).
var circle1 = new Circle("circle1", 100);
var circle2 = new Circle("circle2", 200);
var square = new Square("my square", 150);

window.onload = function() {
    addShapeToPage(circle1);
    addShapeToPage(circle2);
    addShapeToPage(square);
};

function addShapeToPage(shape) {
    var container = document.getElementById("container");
    var div = document.createElement("div");
    var width = 0;
    var classes = "shape ";
    if (shape instanceof Circle) {
        classes += "circle";
        width = shape.radius;
    } else if (shape instanceof Square) {
        classes += "square";
        width = shape.size;
    }
    div.setAttribute("class", classes);
    div.style.left = Math.floor(Math.random() * (container.offsetWidth - 175)) +
    "px";
    div.style.top = Math.floor(Math.random() * (container.offsetHeight - 175)) +
    "px";
    div.style.width = width + "px";
    div.style.height = width + "px";
}
```

```

var span = document.createElement("span");
span.innerHTML = shape.getName();
span.style.visibility = "hidden";
div.appendChild(span);

div.onmouseover = function() {
    // this is the div (the shape) you click on
    this.firstChild.style.visibility = "visible";
};

container.appendChild(div);
}

```



and **Preview**. The program works as it did before; you see three shapes (two circles and a square) in the page and when you mouse over the shapes, their names appear inside of them.

Open up the console and type this:

#### INTERACTIVE SESSION:

```

> circle1.constructor.prototype
Circle { getCircumference: function, getName: function }

```

Now the *prototype* of **circle1** contains the two methods, **getCircumference()** and **getName()**. Check **circle2** in the same way. Take a look at the **Square.prototype** object as well; it contains two methods now. So what did we do?

First, we removed the **getCircumference()** and **getName()** methods from the **Circle()** constructor (we also removed the **getArea()** and **getName()** methods from the **Square()** constructor). Then we added these methods to the **Circle.prototype** object, by setting the same property names (the method names) to the functions (we treat the **Square** prototype similarly):

#### OBSERVE:

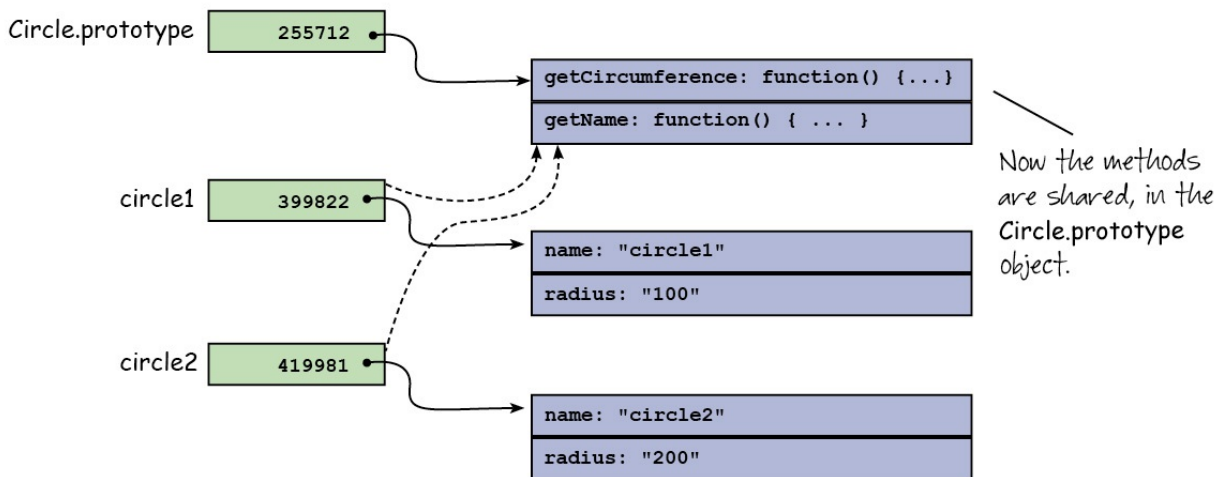
```

Circle.prototype.getCircumference = function() {
    return this.radius * Math.PI * 2;
};
Circle.prototype.getName = function() {
    return this.name;
};

Square.prototype.getArea = function() {
    return this.size * this.size;
};
Square.prototype.getName = function() {
    return this.name;
};

```

The **Circle.prototype** and **Square.prototype** objects, like any other object, can have methods. In the same way that you can add new properties to an object at anytime, because objects are dynamic, we can add new properties to the **Circle.prototype** and **Square.prototype** objects after those prototype objects have been created. So now, the **getCircumference()** and **getName()** methods are stored only once—in the **Circle.prototype** object:



## The Prototype Chain

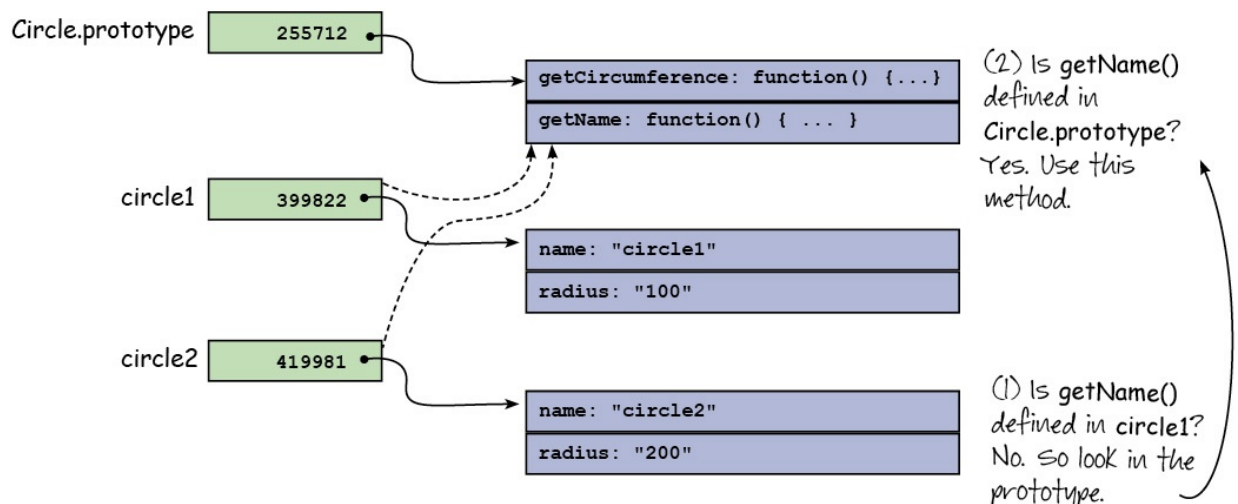
The `getCircumference()` and `getName()` methods are now stored in the `Circle.prototype` object instead of in both the `circle1` and `circle2` objects. Let's see what happens when we try to call one of these methods:

```

INTERACTIVE SESSION:

> circle1.getName()
"circle1"
> circle1.getCircumference()
628.3185307179587
  
```

We called the `getName()` method on the `circle1` object, just like we did before, but `circle1` no longer contains that method. Instead, that method is now in `circle1`'s prototype object. So, how does it work? JavaScript uses the **prototype chain** to look for a property. If a property is not found in an object, JavaScript looks at that object's prototype to see if it's there:



The `getName()` method *is* in the prototype, so that method is called.

Notice that the method in the prototype object still uses `this`. So how is `this` being set to the correct object? After all, the `getName()` method is now in the prototype object, but it still works for both circle objects.

When you call a method, like `circle1.getName()`, the object that contains the method being called, in this case `circle1`, is used as `this`, even if that method is in the object's prototype. So, when you call `circle1.getName()`, `this` is set to `circle1`. When you call `circle2.getName()`, `this` is set to `circle2`.

Let's go back to the console and do a little more testing to help all this sink in (this next session is from the

Chrome console, so use Chrome if you want to duplicate it exactly):

```
INTERACTIVE SESSION:

> Circle
function Circle(name, radius) {
  this.name = name;
  this.radius = radius;
}
> Circle.prototype
Circle { getCircumference: function, getName: function }
> circle1
Circle { name: "circle1", radius: 100, getCircumference: function, getName: function }
> circle1.constructor
function Circle(name, radius) {
  this.name = name;
  this.radius = radius;
}
> circle1.constructor.prototype
Circle { getCircumference: function, getName: function }
> circle2.constructor.prototype
Circle { getCircumference: function, getName: function }
```

First, we check the value of **Circle**. It's just the constructor function **Circle()**. Then we check the value of **Circle.prototype**. Remember, functions are objects, so they can have properties, just like any other object. We ask for the value of the **prototype** property of the **Circle()** constructor function; the value is a **Circle** object containing two properties **getCircumference()** and **getName()**, both of which are methods.

Next, we check the value of **circle1**. This is a **Circle** object, because it was constructed with the **Circle()** constructor function, and contains four properties: **name**, **radius**, **getCircumference()** and **getName()**. But wait! We moved the two methods to the prototype. Why are they listed here in the object? In this case, it's because the Chrome console is showing the properties in the **circle1**'s prototype object, to demonstrate that they are accessible as properties. (We'll see soon how to tell whether a property exists in an object or an object's prototype.)

Finally, we check the prototype of both the **circle1** and **circle2** objects by first getting the **constructor** (using the **constructor** property of the object, which is **Circle()**), and then getting the prototype, using the **prototype** property of the constructor. We see that the prototype of both these objects is a **Circle** object.

Try this with the **square** object.

We are accessing the **getName()** method of the object that's in the variable **shape** when we create the `<span>` that goes inside the `<div>` representing the shape:

```
OBSERVE:

var span = document.createElement("span");
span.innerHTML = shape.getName();
span.style.visibility = "hidden";
div.appendChild(span);
```

Just like we saw in the console session, when we access **shape.getName()**, we're using the method that's stored in the shape's prototype. It doesn't matter if the shape is **circle1**, **circle2**, or **square**; we find the **getName()** in the prototype (**Circle.prototype** or **Square.prototype**, depending on which type of object is stored in the variable **shape**), and **this** gets set to the object in the variable **shape**.

Since **Circle.prototype** and **Square.prototype** are objects, they also have prototypes. The prototype of **Circle.prototype** is **Object.prototype**. Remember that **Object()** is a built-in constructor function that is used to create objects, like when you write:

## INTERACTIVE SESSION:

```
> var o = { x: 3 };
undefined
> o
Object { x: 3 }
> o.constructor
function Object() { [native code] }
> Object.prototype
Object { }
```

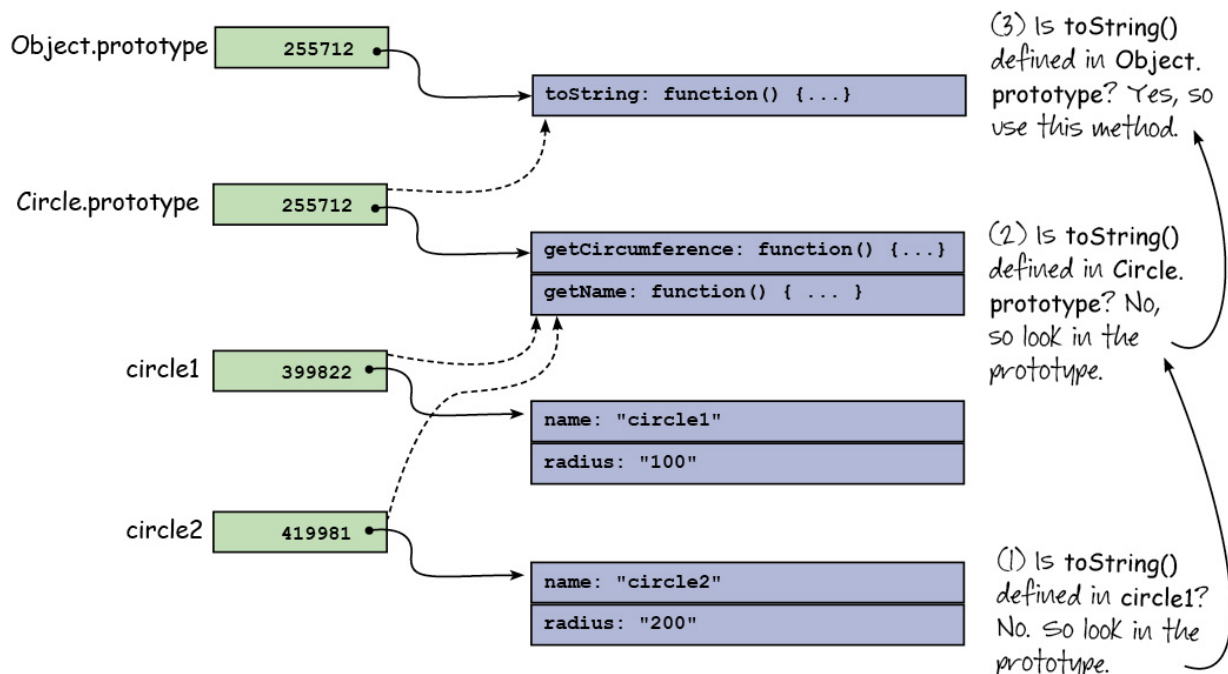
`o` is an object, and because it's a literal object, its constructor is `Object()`. The `Object()` constructor has a **prototype** property, which contains `o`'s prototype. `Object.prototype` looks like an empty Object, but it only *appears* that way in the console. In reality, this object contains a bunch of built-in methods for objects, like `toString()`. You can't see them because the implementation of `Object.prototype` is internal to the browser (that is, it's not JavaScript you wrote); it's just like the [native code] you see when you try to inspect `Object()`.

So let's see what happens when we type this:

## INTERACTIVE SESSION:

```
> circle1.toString();
"[object Object]"
```

Remember the prototype chain: when you write `circle1.toString()`, JavaScript looks first in `circle1`, doesn't find the `toString()` method there, so then it looks in `circle1`'s prototype (`Circle.prototype`). If it doesn't find the method there, it looks in `circle1`'s prototype's prototype (`Object.prototype`) and finds the method there. We call this "looking up the prototype chain" to find the method:



So what's the prototype of the object `Object.prototype`? It's the only object in JavaScript that doesn't have a prototype. `Object.prototype` is the *top* of the prototype chain, so the lookup ends there.

## Prototypal Inheritance

Every object you create **inherits** properties from the prototypes all the way up the prototype chain. In our example, `circle1`, inherits the methods `getName()` and `getCircleCircumference()` from its prototype (because we specifically added them to the prototype), and `circle1` inherits the method `toString()` from the

## Object.prototype.

We call this **prototypical inheritance**, which means that if you try to access a property in an object, and that property doesn't exist in the object itself, JavaScript looks up the prototype chain to try to find that property. Most objects either have one or two prototypes. An object will have the prototype, **Object.prototype**, if it is an object literal, or if it has been created using **new Object()**. An object will have two prototypes if it is created using a constructor function like **Circle()**, in which case the first prototype up the chain is the **Circle.prototype** and the second is one more step up the chain, **Object.prototype**. Some objects have a longer chain, but that's fairly rare.

Let's come back to **instanceof** for a moment. Earlier we talked about **instanceof** as an operator to determine which type of object you have. We use it in the example for this lesson to figure out whether **shape** is a circle or a square:

### OBSERVE:

```
if (shape instanceof Circle) {
  classes += "circle";
  width = shape.radius;
} else if (shape instanceof Square) {
  classes += "square";
  width = shape.size;
}
```

**instanceof** checks the prototype chain of **shape** to see if there is a **Circle.prototype** object or a **Square.prototype** object. In our example, the **shape** was constructed by either the **Circle()** or **Square()** constructor functions, so our **shape** will have a **Circle.prototype** prototype, or a **Square.prototype** prototype, and so we'll find one of these prototypes in the chain.

Remember, **Object.prototype** is *also* in the prototype chain, so we could write this:

### INTERACTIVE SESSION:

```
> circle1 instanceof Circle
true
> circle1 instanceof Object
true
> square instanceof Square
true
> square instanceof Object
true
```

**circle1** is an instance of *both* a Circle *and* an Object! The same is true with **square**; **square** is an instance of *both* a Square and an Object. That makes sense, because *all* objects are instances of Object, after all. However, the extra object in the prototype chain gives you more information about the kind of object you're dealing with. In this case, it tells you whether the object is a circle or a square.

If you have experience using an object-oriented language like Java or C++ or C#, *prototypical inheritance* probably seems quite strange to you. It certainly is different from *classical* inheritance, where you create objects from classes, and create an inheritance hierarchy by *extending* other objects. For more on prototypical inheritance and classical inheritance, and a comparison of the two, see the [Wikipedia page on Prototype-based programming](#).

## When are Prototype Objects Created?

You can add properties to prototype objects, like **Circle.prototype**, after these objects are created, but when *are* prototype objects created? A prototype object is created whenever you define a function. In other words, every function has a prototype property that contains a prototype object. If you then use that function to create a new object using **new**, that new object gets that function's prototype.

So we can add properties to the **Circle.prototype** and **Square.prototype** objects as soon as the **Circle()** and **Square()** functions have been defined, even before we use them to create any objects. Look back at the code and you'll see that we add properties to both prototypes, before we ever define **circle1**, **circle2**, or **square**.

Try this in the console (make sure you've loaded the **proto.html** document first):

#### INTERACTIVE SESSION:

```
> Circle.prototype.x = 400;
400
> Circle.prototype.y = 200;
200
> circle1.x
400
> circle1.y
200
> circle2.x
400
> circle2.y
200
```

We added two properties to the **Circle.prototype** object *after* we created the **circle1** and **circle2** objects, and yet you can see that those properties have values when we query **circle1** and **circle2**. It may seem odd that properties can appear in objects after they're created, without modifying those objects specifically (by adding a property directly, like **circle1.x = 400**);. If you look back at the diagram that shows how prototypes work though, you'll see that if you try to access a property in an object and that property doesn't exist in the object itself, we go up the prototype chain to find it. Even if we add a property to the prototype of an object *after* that object is created, that property will be accessible from the object.

## hasOwnProperty

An object can inherit properties from its prototype chain, but how can you find out if a property is in the object itself, or in one of the object's prototypes? You use the **hasOwnProperty()** method. Make sure your **proto.html** document is loaded and try this in the console:

#### INTERACTIVE SESSION:

```
> circle1.hasOwnProperty("name")
true
> circle1.hasOwnProperty("getName")
false
> circle1.hasOwnProperty("radius")
true
> Circle.prototype.x = 400;
400
> circle1.hasOwnProperty("x")
false
```

The property name you pass to **hasOwnProperty()** must be a string (put the property name in quotation marks). The result of calling **circle1.hasOwnProperty("name")** is true because the property **name**, is defined in the **circle1** object. However, the **getName** property (a method) is *not* defined in the **circle1** object, it's defined in **circle1**'s prototype, **Circle.prototype**, so the result of calling **hasOwnProperty()** on this property is false. Similarly, the property **radius** is defined in **circle1**, but the property **x** is not; it's defined in the **Circle.prototype**.

Try this:

#### INTERACTIVE SESSION:

```
> circle1.hasOwnProperty("hasOwnProperty");
false
> Object.prototype.hasOwnProperty("hasOwnProperty");
true
```

We called the **hasOwnProperty()** method on the **circle1** object, but **hasOwnProperty()** isn't defined in either **circle1** or **Circle.prototype**, so it must be defined in **Object.prototype**. Since *all* objects inherit

this method, you can call **hasOwnProperty()** on any object, including the **Object.prototype** object, which contains this method. That's kind of weird, but you can see why the result of calling `Object.prototype.hasOwnProperty("hasOwnProperty");` is true.

## \_\_proto\_\_

You've seen the `__proto__` property used in a previous lesson when we inspected an object. You can see it if you look at **circle1** in the Chrome console and expand the object by clicking on the little arrow next to it:

```
> circle1
  ▼ Circle {name: "circle1", radius: 100, getCircumference: function, getName: function, x: 400...} ⓘ
    name: "circle1"
    radius: 100
    __proto__: Circle
```

This property refers to the prototype of an object. In this case, our object is **circle1**, and its prototype is **Circle.prototype**, displayed as **Circle** in the `__proto__` property. In fact, in the Chrome console, you can ask for the value of that property, like this:

```
> circle1.__proto__
  ▼ Circle {getCircumference: function, getName: function, x: 400, y: 200} ⓘ
    ▶ constructor: function Circle(name, radius) {
    ▶ getCircumference: function () {
    ▶ getName: function () {
      x: 400
      y: 200
    ▶ __proto__: Object
  > |
```

The prototype of **circle1** is a **Circle** object that contains the two methods we added, **getName()** and **getCircumference()**, along with the properties **x** and **y** that we added to the prototype later (you might not see **x** and **y** if you've reloaded the page). **Circle.prototype** has a `__proto__` property, **Object.prototype**, so the `__proto__` property of objects allows you to see the prototype chain.

However, **do not rely on this property**. It's kind of a secret property that browsers implement, but it's not officially part of the JavaScript standard and could disappear or change at any time. Use it to inspect objects in the console, but *not* in your programs!

## Setting the Prototype Property to an Object Yourself

So far, we've used the default prototype you get when you create a constructor function, and then added our own properties and methods to that prototype object for inheritance, but you can actually set the prototype of a constructor yourself to an object you've created. Let's see how:



## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Shapes with Prototypes and Inheritance: Setting the prototype yourself
</title>
  <meta charset="utf-8">
  <script>
    var shape = {
      x: 0,
      y: 0,
      area: 0,
      setPosition: function(x, y) {
        this.x = x;
        this.y = y;
      },
      displayInfo: function() {
        console.log("Your shape has area " + Math.ceil(this.area) + ", and is
located at " + this.x + ", " + this.y);
      }
    };

    function Circle(radius) {
      this.radius = radius;
      this.computeArea = function() {
        this.area = Math.PI * (this.radius * this.radius);
      };
    }


    function Square(size) {
      this.size = size;
      this.computeArea = function() {
        this.area = this.size * this.size;
      };
    }

    Circle.prototype = shape;
    Square.prototype = shape;

    var circle = new Circle(50);
    circle.setPosition(100, 100);
    circle.computeArea();
    circle.displayInfo();

    var square = new Square(50);
    square.setPosition(300, 300);
    square.computeArea();
    square.displayInfo();
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **setProto.html**, and [Preview](#) . Open the console, and you see this:

## INTERACTIVE SESSION:

```
Your shape has area 7854, and is located at 100, 100
Your shape has area 2500, and is located at 300, 300
```

We defined a literal object, **shape**, using some properties and methods. We made this a literal object, not a

constructor because we don't want users to create new shape objects; we want them to create circles and squares.

Then we define two simple constructors, one for **Circle** objects, and one for **Square** objects:

```
OBSERVE:
Circle.prototype = shape;
Square.prototype = shape;
```

We **set the prototype property** of the **Circle** constructor object to the **shape** object, and **we do the same for the Square constructor object**. So now, the prototype object for **circle** is a **shape** object rather than a **Circle**. As such, **circle** inherits the methods and properties of the **shape** object, so we can call **circle.setPosition()** and **circle.displayInfo()**. **square** works the same way as well.

In this example, we use the *same* object as the prototype for both circles and squares: **shape**. That means the properties and methods in the prototype object need to make sense for both circles and squares. Before, we set properties of the prototype by writing **Circle.prototype.PROPERTY = PROPERTY VALUE** and **Square.prototype.PROPERTY = PROPERTY VALUE** because we needed different properties and methods for circles and squares. Keep this in mind as you are setting up your object prototypes; how you choose to set up the prototype chain depends on your individual situation and whether you need different prototypes or the same prototype for your objects.

Take a look at the **circle** and **square** objects:

```
INTERACTIVE SESSION:
> circle
Circle {radius: 50, computeArea: function, x: 100, y: 100, area: 7853.981633974483}
> square
Square {size: 50, computeArea: function, x: 300, y: 300, area: 2500}
> circle.constructor.prototype
Object {}
> square.constructor.prototype
Object {}

> circle instanceof Circle
true
> circle instanceof shape
TypeError: Expecting a function in instanceof check, but got #<Circle>
```

We see that the constructor for **circle** is **Circle**, and for **square**, it's **Square**, but now, the **circle.constructor.prototype** is shown as **Object {}** (it was **Circle {}** before). That's because the prototype is the **square** object, and since the **square** object is a literal object, its constructor is **Object()**, so its "type" is **Object**. **square** works the same way.

In other words, **circle** is an instance of **Circle** (because we made **circle** using the **Circle()** constructor), and **circle** is an instance of **Object** because **shape** is a literal object, and its constructor is **Object()**.

Prototypal inheritance and object prototypes are not extremely complicated, yet this topic can be difficult to wrap your head around, especially if you have experience with class-based languages. You have to put all that knowledge aside and think differently about objects and inheritance. It all boils down to this: in Javascript, every object has a prototype (except, of course, **Object.prototype**), and can inherit properties from the prototype chain.

Make sure you give yourself plenty of time to practice and understand the concepts in this lesson. Experiment with your own objects and prototypes. Use the console to inspect your objects, and the tools you have in your pocket now to test to make sure that what you think should happen does. Practice and make sure you know what you're doing in the quizzes and projects.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Functions

---

## Lesson Objectives

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

- use basic functions.
  - use functions to organize code.
  - define functions using function declarations and function expressions.
  - compare the differences in using function declarations and function expressions.
  - express functions as first class values.
  - express functions as anonymous functions.
  - pass functions as values to other functions.
  - return a function from a function.
  - use a function as an event handler.
  - recognize how values are passed to functions using pass-by-value.
  - explain what happens when we pass objects to functions.
- 

We've spent the last couple of lessons working with objects; now let's turn our attention to functions.

## JavaScript Functions

We've used functions throughout the course, in a variety of different ways. It would be difficult to do any kind of serious JavaScript without using a function.

### What is a Function?

A function is a block of code that's defined once, but can be executed many times. Let's look at a simple function declaration:

OBSERVE:

```
function computeArea(radius) {  
    var area = radius * radius * Math.PI;  
    return area;  
}
```

There are a lot of different parts to this function, so let's go through it, step by step. First, we have the **function** keyword. Then, we have the **name of the function, computeArea**. Then in parentheses, we have a **parameter (radius)**. The body of the function (the statements that are executed when you call the function) are defined within curly brackets. In the body of this function, we have two statements: first a statement that declares and computes a value for a **local variable, area**, second a **return statement**, that returns the value of **area** to the statement that called the function.

Let's call the function now:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    function computeArea(radius) {
      var area = radius * radius * Math.PI;
      return area;
    }

    var circleArea = computeArea(3);
    console.log("Area of circle with radius 3: " + circleArea);
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **functions.html**, and **Preview** . In the console, you see the area of the circle with the radius 3:

**OBSERVE:**

```
Area of circle with radius 3: 28.274333882308138
>
```

Let's take a closer look:

**OBSERVE:**

```
function computeArea(radius) {
  var area = radius * radius * Math.PI;
  return area;
}

var circleArea = computeArea(3);
```

We call the function using its **name (computeArea)**, passing an **argument (3)** to the function. The function's parameter, **radius** gets the value of the argument. That value is used in the computation. When we pass an argument into a function, we say that the argument is **bound** to the parameter. So here, the value 3 is bound to the parameter **radius**.

The value returned from the function is stored in the variable **circleArea**.


We can call the function as many times as we want:

**CODE TO TYPE:**

```
function computeArea(radius) {
  var area = radius * radius * Math.PI;
  return area;
}

var circleArea = computeArea(3);
console.log("Area of circle with radius 3: " + circleArea);
circleArea = computeArea(5);
console.log("Area of circle with radius 5: " + circleArea);
circleArea = computeArea(7);
console.log("Area of circle with radius 7: " + circleArea);
```



and **Preview** . In the console, you now see the areas of the circles with radii of 5 and 7, along with the

area of the circle with radius 3.

By creating the function **computeArea()**, we packaged up a little bit of code that computes the area of a circle; that can then be reused each time we call the function. In addition, we can customize the code a bit each time we call the function by passing different values for the argument. We can also get customized values back from a function if we return a value. In this case, we get the area of the circle that provided the radius we passed into the function. Note that functions always return a value; if you don't explicitly return one, a function returns **undefined** (unless you use it as a constructor, in which case the function returns an object).

So, functions are a way of reusing code. They're also a way of organizing your code. A good programming practice is to think of functions as a way to put related code together. For instance, you might put all code related to computing the area of a circle together in one function, while you put all code related to computing the distance between two points together in another function.

When you create a function, you're also creating a **scope** for executable statements. We'll look at scope in a lot more detail in the next lesson, but for now, notice that the parameter **radius** and the variable **area** are both **local** (that is, visible only in the body of the function), rather than **global** (that is, visible everywhere in your code). Programmers are often critical of JavaScript's dependence on global variables, because global variables are easy to lose and misuse. Functions are good for keeping variables out of the global scope. This is especially useful when you combine your own code with code from libraries, like jQuery or Underscore.js. Using functions to keep variables out of the global scope is often called the **Module Pattern**. We'll cover that in detail in a later lesson.

## Different Ways of Defining a Function

In the code above, we used what's called a **function declaration** to define the **computeArea()** function. That is, we declared the function using a statement that begins with the **function** keyword. It looks like this:

OBSERVE:

```
function functionName(parameters) {  
  // body goes here  
}
```

One of the advantages to defining functions using function declarations is that you can place your functions above or below the code that uses them. Try it:

CODE TO TYPE:

```
function computeArea(radius) {  
  var area = radius * radius * Math.PI;  
  return area;  
}  
  
+  
  
var circleArea = computeArea(3);  
console.log("Area of circle with radius 3: " + circleArea);  
circleArea = computeArea(5);  
console.log("Area of circle with radius 5: " + circleArea);  
circleArea = computeArea(7);  
console.log("Area of circle with radius 7: " + circleArea);  
  
function computeArea(radius) {  
  var area = radius * radius * Math.PI;  
  return area;  
}
```



and **Preview**; your code works exactly the same way as it did before, even though the function **computeArea()** is defined *below* where we are using it.

This works because when the browser loads your page, it goes through all your JavaScript and looks for function declarations *before* it begins executing your code. When you define a function at the global level like we did here, JavaScript adds the function as a property of the global window object, so that the function definition is visible everywhere in your code. Then, the browser goes back to the top of your JavaScript, and begins executing the code, top down. So, when the JavaScript interpreter gets to the first line where you call **computeArea()**, that function is defined, so the function call succeeds.



Another way you can create a function is to use a *function expression*:

#### CODE TO TYPE:

```
var circleArea = computeArea(3);
console.log("Area of circle with radius 3: " + circleArea);
circleArea = computeArea(5);
console.log("Area of circle with radius 5: " + circleArea);
circleArea = computeArea(7);
console.log("Area of circle with radius 7: " + circleArea);

function computeArea(radius) {
  var area = radius * radius * Math.PI;
  return area;
}
+
var computeArea = function(radius) {
  var area = radius * radius * Math.PI;
  return area;
};
```

We've replaced the function declaration with a variable declaration: we declare the variable **computeArea** and initialize that variable to the result of a **function expression**, that is, a function value. Because **computeArea** is a global variable, the end result is almost the same: a property named **computeArea** is

added to the global window object set to the value of the function. However, when you  and **Preview**  preview, you see an error instead of the expected log messages. Why?



JavaScript sees this statement as just a variable declaration and initialization. The value of the variable happens to be a function, but because we are not using a function declaration, the function is no longer defined in that first pass through the code; instead, the function is not defined until JavaScript gets to the variable declaration, which is when it executes the code from the top down. Now that the function is defined *after* the statements that try to call the function, we get an error message.

We can fix the error by moving the variable declaration to the top of the code, like this:

#### CODE TO TYPE:

```
var computeArea = function(radius) {
  var area = radius * radius * Math.PI;
  return area;
};
var circleArea = computeArea(3);
console.log("Area of circle with radius 3: " + circleArea);
circleArea = computeArea(5);
console.log("Area of circle with radius 5: " + circleArea);
circleArea = computeArea(7);
console.log("Area of circle with radius 7: " + circleArea);

var computeArea = function(radius) {
  var area = radius * radius * Math.PI;
  return area;
}
+
+;
```

 and **Preview** ; your code works again.

We also use function expressions when we define methods in objects:

#### OBSERVE:

```
circle = {
  radius: 3,
  computeArea: function() {
    var area = this.radius * this.radius * Math.PI;
    return area;
  }
};
```

You can see we're using the same syntax to define the method (**computeArea()** in the **circle** object) as we did when we defined it as a global function. The difference is that now the property is visible only in the **circle** object; the function is no longer a property of the global **window** object.

So, when defining global functions, which is better: using a function declaration, or declaring a variable and initializing it to a function expression?

The main advantage to using function declarations when defining *global* functions is that you know the functions will be visible throughout your code so you don't necessarily need to put them all at the top. However, as long as you don't need all of your functions to be defined at the time your code begins executing, using a variable declaration and setting the value of the variable to a function expression works just as well. Some programmers prefer that method of creating functions.

We often create and use functions that don't need to be defined in the global window object. Reducing the number of global variables (including functions!) is always preferable in JavaScript. So next, we'll take a look at other situations where we can use function expressions rather than function declarations.

## Functions as First Class Values

We tend to think of functions as different from other kinds of values, like 3, or even an object, like **circle**, but in JavaScript, a function is just another kind of value, a value that you can assign to a variable or an object property, and even pass to or return from a function.

We say that functions are **first class values** in JavaScript. A first class value is one that can be stored in a variable, passed to a function, and returned from a function. You can already see that values like numbers, strings, and booleans are first class; so are objects. If you've been working with JavaScript for a while, you probably know that functions are first class. Not all languages have first class functions. In some languages, functions are treated separately and differently than other values.

So what's the deal with **first class functions**? We'll take a look at the **map()** method to answer that question. Create a new file and add this code:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> First Class Functions </title>
  <meta charset="utf-8">
  <script>
    var myArray = [1, 2, 3];
    var newArray = myArray.map(function(x) { return x+1; });
    console.log(newArray);
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **functions2.html**, and . Open the console. You see an array in the console:

#### OBSERVE:

```
[2, 3, 4]
```

**Note**

If you get an error when you try this code, it's because your browser doesn't yet support **map()**. This is a fairly new method that was added to JavaScript as part of the ECMAScript 5 standard. Make sure you have the most recent version of your browser to try this code, as **map()** has broad support in all the recent versions of browsers.

The **map()** method is an array method that all arrays inherit from the Array prototype. It takes a function and applies that function to each element of the array (in order). So in our example, **map()** first applies the function to **myArray[0]**, then **myArray[1]**, and so on. The array element is passed as the argument for the parameter **x**. **map()** returns a new array, the same length as the original array, with elements that are the values returned by each invocation of the function we passed to **map()**. In our example, the function we pass to **map()** adds one to each of the array elements, so the array you get back has items that are one greater than each of the corresponding items in the original array.

The value that we passed to **map()** is a function. Unfortunately, since **map()** applies the function to the elements of the array for you (behind the scenes), you don't get to see how a function that is passed as an argument works. Let's implement our own version of **map()**, that way, you can see not only how to pass a function as an argument, but also how to use it in the function to which you pass it:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> First Class Functions </title>
  <meta charset="utf-8">
  <script>
    var myArray = [1, 2, 3];
    var newArray = myArray.map(function(x) { return x+1; });
    console.log(newArray);

    function map(a, f) {
      var newArray = [];
      for (var i = 0; i < a.length; i++) {
        newArray.push(f(a[i]));
      }
      return newArray;
    }

    function addOne(x) {
      return x+1;
    }

    var newArray2 = map(myArray, addOne);
    console.log(newArray2);
  </script>
</head>
<body>
</body>
</html>
```



and **Preview** preview. In the console, you get the exact same result for our own version of **map()** as we did before, the array [2, 3, 4]. Let's go over the code, step by step:



OBSERVE:

```
function map(a, f) {
  var newArray = [];
  for (var i = 0; i < a.length; i++) {
    newArray.push(f(a[i]));
  }
  return newArray;
}

function addOne(x) {
  return x+1;
}

var newArray2 = map(myArray, addOne);
```

Our version of `map()` is a function, not a method, so we need to pass both the **array** and the **function** that will act on the array, to the `map()` function. That's why our function has two parameters instead of one.

First, we create a new empty array, `newArray`. Then we loop over all the elements in the array we passed in, `a`, and apply the function `f` to the array element. Inside `map()`, we find `f`. `f` produces a new value that we then add to the `newArray` at the same index as `a[i]`. When we finish, we return the `newArray`.

Now let's see how to call this `map()` function. We need an **array** and a **function** to pass to it; we'll reuse `myArray` (from the top of the code), and create a function named `addOne()` to pass. Just like before, `addOne()` is a function that takes one argument, adds one to it, and returns that new value. We call `map()`, passing in `myArray` and `addOne` as arguments for `a` and `f`, then get back a new array with each element one greater than `myArray`.

Passing a function to a function is much like passing any other value to a function, except you have to use it as a function, and you need to know what kind of arguments the function expects and what kind of value it returns (if any). You can write methods and functions like `map()`, that do some useful work, and can also be customized by passing in different functions.

## Anonymous Functions

When we called the `map()` method on the array, we used a function expression as the argument that we passed to `map()`, whereas in our own implementation of `map()`, we use a declared function as the argument. It doesn't really matter which you use, but if you're not going to use the function `addOne()` anywhere other than as an argument to `map()`, you might want to avoid excess clutter and use a function expression instead:

## CODE TO TYPE:


```
<!doctype html>
<html>
<head>
  <title> First Class Functions </title>
  <meta charset="utf-8">
  <script>
    var myArray = [1, 2, 3];
    var newArray = myArray.map(function(x) { return x+1; });
    console.log(newArray);

    function map(a, f) {
      var newArray = [];
      for (var i = 0; i < a.length; i++) {
        newArray.push(f(a[i]));
      }
      return newArray;
    }

    function addOne(x) {
      return x+1;
    }

    var newArray2 = map(myArray, addOne);
    var newArray2 = map(myArray, function(x) { return x+1; });
    console.log(newArray2);
  </script>
</head>
<body>
</body>
</html>
```



and . You see the same result in the console.

Now, instead of declaring the function **addOne()**, we pass a function expression to **map()**. Unlike when we passed **addOne()**, this function doesn't have a name. It's known as an **anonymous function**. Of course, it has a name inside **map()**, because it gets bound to the name of the parameter, **f**.

Anonymous function expressions are just function values where the function has no name. This is useful when we're passing functions to functions, or returning functions from functions, because in both cases the function gets bound to a name so you can refer to it. When you pass an anonymous function to a function, it gets bound to the name of the parameter variable; when you return an anonymous function from a function, it gets stored in the variable you're using to hold the return value. You'll see anonymous functions used frequently in JavaScript as a shortcut to declare and name a function separately.

## Returning a Function from a Function

Not only can you pass a function to a function, you can return a function from a function. Let's look at an example:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Returning Functions </title>
  <meta charset="utf-8">
  <script>
    function makeConverterFunction(multiplier, term) {
      return function(input) {
        var convertedValue = input * multiplier;
        convertedValue = convertedValue.toFixed(2);
        return convertedValue + " " + term;
      };
    }

    var kilometersToMiles = makeConverterFunction(0.6214, "miles");
    console.log("10 km is " + kilometersToMiles(10));

    var milesToKilometers = makeConverterFunction(1.62, "km");
    console.log("10 miles is " + milesToKilometers(10));
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **functions3.html**, and . In the console, you see this output:

#### OBSERVE:

```
10 km is 6.21 miles
10 miles is 16.20 km
```

Let's go over the code:

#### OBSERVE:

```
function makeConverterFunction(multiplier, term) {
  return function(input) {
    var convertedValue = input * multiplier;
    convertedValue = convertedValue.toFixed(2);
    return convertedValue + " " + term;
  };
}

var kilometersToMiles = makeConverterFunction(0.6214, "miles");
console.log("10 km is " + kilometersToMiles(10));

var milesToKilometers = makeConverterFunction(1.62, "km");
console.log("10 miles is " + milesToKilometers(10));
```

We're using one function, **makeConverterFunction()**, to create two other functions, **kilometersToMiles** and **milesToKilometers**. **makeConverterFunction()** takes two arguments: a multiplier value to do a conversion (it doesn't matter what kind of conversion, as long as it can be done by multiplying one value by another), and a string representing the term of measurement we expect back from the function we generate.

**makeConverterFunction()** returns a function. The function it returns takes one argument, **input**, and uses that argument in a computation with the parameters of **makeConverterFunction()**.

**makeConverterFunction()** knows nothing about the kind of conversion we want to do. It knows that it's generating a new function that multiplies **input** by **multiplier**, uses **toFixed()** to make sure the resulting number has a fractional part of at most two numbers, and then returns a string made by combining the number with the **term** passed into **makeConverterFunction()**.

We can use **makeConverterFunction()** to make functions that do a specific kind of conversion, passing in

a value for multiplier and a string for term. So to create a **function that converts kilometers to miles (kilometersToMiles)**, we pass in a multiplier of 0.6214, and "miles," and get back a function that can do this conversion when we call it and pass in the number of kilometers. Similarly, we can create a **function to convert miles to kilometers (milesToKilometers)** by passing in 1.62 for the multiplier, and "km" for the term.

Review this code carefully to make sure you understand it. The **makeConverterFunction()** returns a function that uses both the parameters of **makeConverterFunction()**, as well as the (yet to be bound) parameter, **input**. Once **makeConverterFunction()** creates its function and returns it, the parameters **multiplier** and **term** go away, yet somehow, the functions **kilometersToMiles()** and **milesToKilometers()** "remember" those values. The secret to this somewhat magical ability to remember is the **closure**—we'll come back to that in a later lesson.

For now, just be aware that the arguments you pass into **makeConverterFunction()** are used (and remembered) by the function that **makeConverterFunction()** creates. So in **kilometersToMiles()**, the value of **multiplier** is 0.6214 and the value of **term** is "miles," while in **milesToKilometers()**, the value of **multiplier** is 1.62, and the value of **term** is "km."

## Functions as Callbacks

At the heart of just about every JavaScript program are **events**. When you write a web application, you use JavaScript to add interactivity to your page, which means your program needs to respond to events generated by the browser, and events generated by the user.

In JavaScript, we use functions to handle events. We often refer to event handlers as **callbacks** because when an event happens, we "call back" to a function to handle that event. Callbacks aren't always about browser and user events, but often they are.

You've probably used functions to handle events like the page load and button click events. If you've used Ajax (also known as XHR) or Geolocation in your JavaScript programs, you're probably familiar with functions used as callbacks. For instance, with Ajax, you provide a function to "call back" when XMLHttpRequest has loaded data from a file. With Geolocation, you provide a function to "call back" when the browser has located your position. Let's take a quick look at a basic Geolocation example so you can see how we use a function as a callback:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Functions as callbacks </title>
  <meta charset="utf-8">
  <script>
    window.onload = function() {
      if (navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(getLocation);
      }
      else {
        console.log("Sorry, no Geolocation support!");
        return;
      }
    };

    function getLocation(position) {
      var latitude = position.coords.latitude;
      var longitude = position.coords.longitude;
      var div = document.getElementById("container");
      div.innerHTML = "You are at lat: " + latitude + ", long: " + longitude;
    }
  </script>
</head>
<body>
<div id="container"></div>
</body>
</html>
```



Save this in your /AdvJS folder as **functions4.html**, and . Your location will be displayed

in the browser in a moment.

**Note** All modern browsers support Geolocation, but there are many reasons Geolocation can fail. If you're not getting a location, don't worry, just follow along.

We use functions as callbacks in two places in this example: first, we use an anonymous function as the load event handler, by setting the **window.onload** property to the function. That means when the browser triggers the "load" event—that is, when the page is fully loaded—the function that's been stored in the **window.onload** property is called. This happens *asynchronously*. You can't anticipate when the page is loaded; all you know is that when the browser has loaded the page, it will call this function.

Second, we use the **getLocation()** function as a callback for the Geolocation **getCurrentPosition()** method. Again, this is an event handler: a function that is called when a specific event happens—in this case, when the Geolocation object has found your position. This callback happens asynchronously: you can't anticipate how long it's going to take your browser to find your location, you just know that when it does, the browser will call your function back with your position.

Unlike with **window.onload**, we specify this callback by passing the **getLocation()** function to the **getCurrentPosition()** method. The **getCurrentPosition()** method calls **getLocation()** (behind the scenes) as soon as the browser has retrieved your location. In **getLocation()** we have a valid position, so we add the position, as a latitude and longitude, to the web page. (If you get an error retrieving your position, then **getLocation()** won't be called; it's only called if the browser can find you).


## Calling Functions: Pass-by-Value

In this lesson we've seen quite a few examples of functions, and we've passed a variety of different values to functions, including numbers, strings, and other functions. Let's take a closer look at what happens when we pass arguments to functions. Create a new file and add this code:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Functions: Pass by value </title>
  <meta charset="utf-8">
  <script>
    function changeNum(num) {
      num = 3;
    }
    var myNum = 10;
    changeNum(myNum);
    console.log(myNum);
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **functions5.html**, and . Open the console.

You see the value 10. Is that what you expected? When you pass **myNum** to the function **changeNum()**, the parameter **num** gets a *copy* of the value of **myNum**. When you change **num** to 3, you don't change the value of **myNum**.

This process of *copying* a value into a parameter when you pass an argument to a function is called **pass-by-value**.

Now add this code to the program:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Functions: Pass by value </title>
  <meta charset="utf-8">
  <script>
    function changeNum(num) {
      num = 3;
    }
    var myNum = 10;
    changeNum(myNum);
    console.log(myNum);

    function changeObj(obj) {
      obj.x = 3;
      obj.z = 10;
    }
    var foo = {
      x: 0,
      y: 1
    };
    console.log("Before calling changeObj, foo is: ");
    console.log(foo);

    changeObj(foo);
    console.log("After calling changeObj, foo is: ");
    console.log(foo);
  </script>
</head>
<body>
</body>
</html>
```



and **Preview** . You see this output (or something similar; ours is from Chrome):

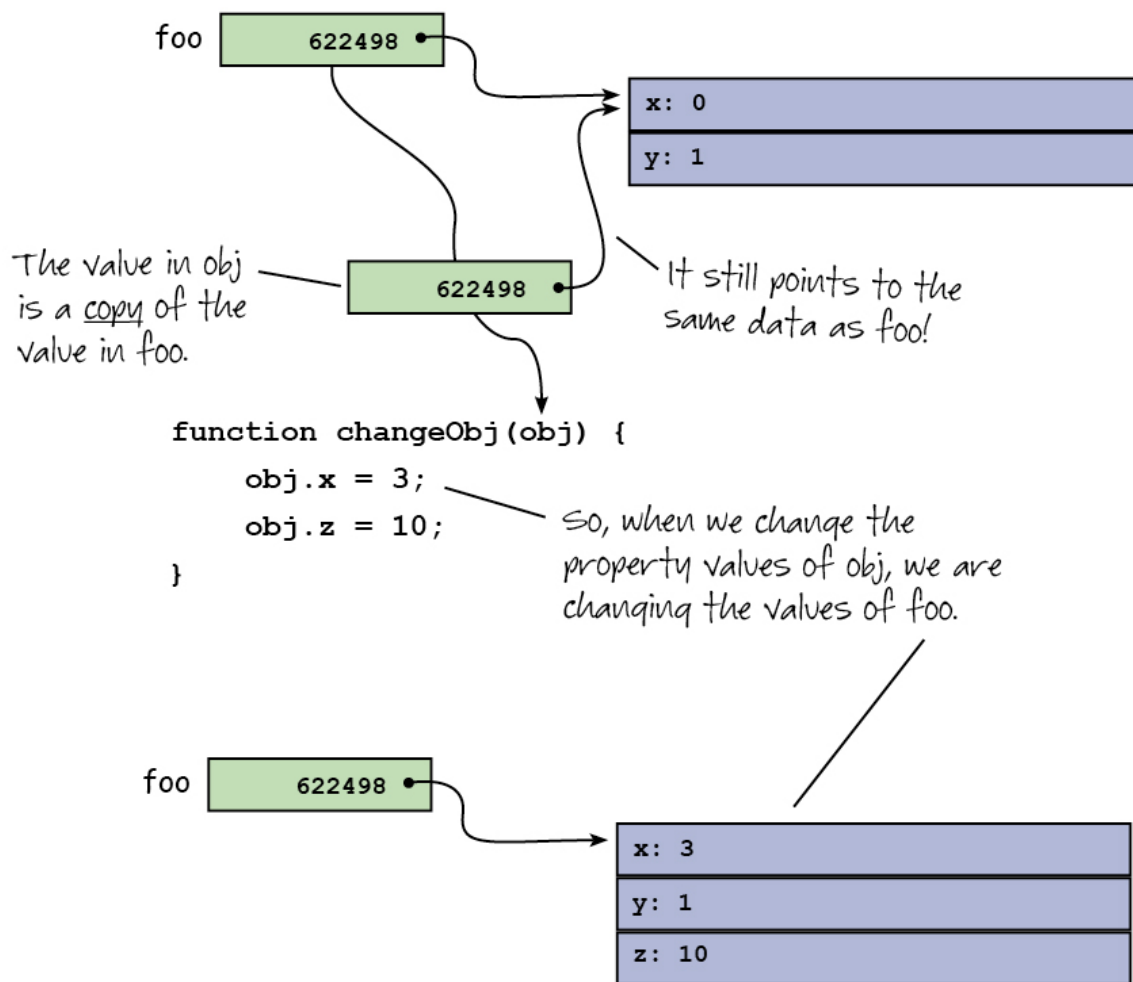
## OBSERVE:

```
Before calling changeObj, foo is:
Object {x: 0, y: 1}
After calling changeObj, foo is:
Object {x: 3, y: 1, z: 10}
```

In the new code, we create an object **foo**, which has two properties: **foo.x** and **foo.y**. We pass the object **foo** to the function **changeObj()**, and the object is bound to the parameter **obj**. The function sets the value of the property **obj.x** to 3, and the value of the property **obj.z** to 10.

After we call **changeObj()**, passing **foo** to the function, the properties of **foo** are different. JavaScript is pass-by-value, which means that function parameters get a *copy* of the value of the arguments we pass to the function. So how are the properties of **foo** being changed?

The value in the variable **foo** is a *reference*—that is, a memory location, a pointer to the data in the object. When we pass **foo** to **changeObj()**, we pass a copy of the *memory location*, not a copy of the data in the memory location. So **obj** (the parameter) is also a memory location, one that points to the *same* place as **foo**. When you change property values, or add new properties to **obj**, you are making those changes and additions to **foo**, because they are pointing to the same object.



This concept of pass-by-value can be a little tricky at first, so spend some time looking over this code to make sure you understand it. Experiment on your own. Try passing an array to a function that changes the array. Remember that arrays are objects too. Do you get the results you expect?

## Return


Before we finish up this lesson, let's talk about the **return** statement. **return** is used to return a value from a function. However, if you don't have a **return** statement, the function still returns a value: **undefined** (unless you're using the function as a constructor, in which case it returns an object).

You might think that return would always be the last statement in the body of your function, but it doesn't have to be. You can use return to exit from a function early. Create a new file and add the code below, so we can see how that works:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    function getWeather(temp) {
      if (temp >= 80) {
        return "It's hot!";
      } else if (temp >= 50 && temp < 80) {
        return "It's nice!";
        console.log("test");
      } else {
        return "It's cold!";
      }
    }
    var returnValue = getWeather(71);
    console.log(returnValue);
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **functions6.html**, and **Preview** . In the console, you see the message "It's nice," but you won't see the message "test." We return a string from the **getWeather()** function, and then display that value in the console. So **returnValue** holds the value returned from the function. Notice that we've got three **return** statements in the function now. As soon as any one of them is executed, the function returns, so any code that follows the return statement will be ignored. When we return the string, "It's nice," the function execution stops, so we never see the message "test" in the console.

When you return a value from a function, write your return statement with care. Change your code just a tiny bit as shown:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Functions </title>
  <meta charset="utf-8">
  <script>
    function getWeather(temp) {
      if (temp >= 80) {
        return "It's hot!";
      } else if (temp >= 50 && temp < 80) {
return "It's nice!";
        return
          "It's nice!";
console.log("test");
      } else {
        return "It's cold!";
      }
    }
    var returnValue = getWeather(71);
    console.log(returnValue);
  </script>
</head>
<body>
</body>
</html>
```



and **Preview** . Now all you see in the console is **undefined**; you don't see the message, "It's nice!"



Usually you can add whitespace in a JavaScript program without affecting the way the program executes, but in this case, the function is no longer working as we expect.

This is an example of "automatic semicolon insertion" and it's a holdover from when people used to write JavaScript without using semicolons. JavaScript reads the above code like this:

OBSERVE:

```
return;  
"It's nice!";
```

JavaScript thinks you forgot a semicolon at the end of the return statement, so it "**helpfully**" **inserts one for you**, and in the process breaks your code. There is no error because "It's nice" is a valid expression and statement (it doesn't do a whole lot, but it's perfectly valid). So, your function returns when it hits the statement **return**; and, because you're returning with no value, the value returned from the function is **undefined**. The statement "**It's nice**"; never gets executed (and even if it did, as a standalone statement, it wouldn't do anything visible).

The **return** statement is not the only situation where JavaScript does automatic semicolon insertion, but it's a common cause of errors, so watch out for it! You can read more about [automatic semicolon insertion in the JavaScript specification \(5.1\)](#).

Take a break to rest your brain, and then tackle the quizzes and projects to digest all of this new information.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Scope

## Lesson Objectives

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

- use local scope and global scope.
- use functions to create local scope.
- recognize variables that are hoisted within a function.
- use nested functions.
- explore lexical scoping within nested functions.
- use a scope chain to recognize how variables are resolved.
- use the Chrome Developer Tools to inspect the scope chain.

## Scope

To truly understand functions, and JavaScript in general, you need to understand scope. JavaScript has two kinds of scope: *global*, meaning a variable is visible everywhere, and *local*, meaning the variable is visible only within a function. You're most likely using JavaScript in the browser, so you know that the global scope comes set up with an object, **window**, which exposes all the browser-related JavaScript features you need to create web applications. In this lesson, we'll dive into scope; we'll take another look at how variables and functions work from the perspective of scope, JavaScript's scoping rules, and how to plan your code to avoid certain "gotchas." We'll also use Chrome's web developer tools to inspect the function *call stack* and get a first-hand look at scope in action. Let's get going!

## Variable Scope

We'll start by looking at the *global scope*. Here are three ways to create a global variable:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Global Scope </title>
  <meta charset="utf-8">
  <script>
    var globalScope1 = "Global";

    //
    // not using var to define a new variable is bad form!
    //
    globalScope2 = "Global";

    function f() {
      globalScope3 = "Global";
    }
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **globalScope.html**, and **Preview** . Open the console, and see which of the variables is defined globally:

#### INTERACTIVE SESSION:

```
> globalScope1
"Global"
> globalScope2
"Global"
> globalScope3
ReferenceError: globalScope3 is not defined
```

Both **globalScope1** and **globalScope2** are global variables, so you can inspect them in the console, but we haven't called the function **f()**, so **globalScope3** hasn't yet been created. Call **f()** and then check again:

#### INTERACTIVE SESSION:

```
> f()
undefined
> globalScope3
"Global"
```

Now, **globalScope3** is defined, and it is a global variable. Why is **globalScope3** a global variable? After all, we defined it inside the function **f()**. Well, notice that we did not use **var** to declare the variable. When you use a new variable in a function without using the keyword **var**, JavaScript automatically creates a new global variable for you. This can be a problem if you're not expecting it! For instance, you could accidentally overwrite the value of an existing global variable if you use the same name and forget to write **var**. So, avoid using new variable names within a function to create global variables. In general, always declare your variables with **var**, whether they are global or local.

The term *global scope* describes the visibility of your variables. Global variables are visible *everywhere* in your JavaScript code, including in external files if you are loading external scripts. So smart JavaScript programmers try to avoid global variables except when they're absolutely necessary.

Global variables are added to the *global object*, which is the **window** object in all current browsers. Type **window** in the console:

#### INTERACTIVE SESSION:

```
> window
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
```

You see the window object and its many methods and properties. In Chrome, Safari, and Firefox, if you click the arrow next to the object in the console, you'll see some familiar methods and properties. Scroll down to the properties starting with lower case "g" and look for the global variables you defined in the code above. You see all three properties there. Scroll back up to the "f"s and you see the function **f()** we defined. Whenever you define a global variable, it's added to the global object as a property.

When you use a global variable or method, whether it's one you define yourself (like the global variables we created above) or properties of the global object (like **alert()**, **console.log()**, or **document**, the document object), you don't have to specify **window.alert()**, or **window.globalScope1**; you can just type the name, for instance, **alert()** or **globalScope1**. The global object is the default scope for all variables.

#### INTERACTIVE SESSION:

```
> window.globalScope1
"Global"
> globalScope1
"Global"
```

Both work, so we usually just leave off the "window" part.

## Function Scope

The other kind of scope in JavaScript is *local scope*. A local scope is created whenever you call a function:

```
CODE TO TYPE:

<!doctype html>
<html>
<head>
  <title> Local Scope </title>
  <meta charset="utf-8">
  <script>
    var message = "Loading...";

    window.onload = function() {
      var message = "Done loading";
      var div = document.getElementById("container");
      div.innerHTML = message;
    }
  </script>
</head>
<body>
  <div id="container"></div>
</body>
</html>
```



Save this in your **/AdvJS** folder as **localScope.html**, and **Preview**. You see the message "Done loading" in the web page. Open the console and try to access the variables:

```
INTERACTIVE SESSION:

> message
"Loading..."
> div
ReferenceError: div is not defined
```

We can access the globally defined variable **message**, but not the locally defined variables **message** and **div**. Plus, we've got two variables with the same name. The **message** that's defined inside the (anonymous) function is a *local* variable, while the **message** defined above the function is a *global* variable. Similarly, the variable **div** is a local variable.

Inside the function, we set the **innerHTML** property of the **div** object to the value of the *local message* variable (the one that is defined in the function) because the local **message** variable *shadows* the global **message** variable. If you use a local variable with the same name as a global variable, the local variable is used when you refer to it within the same function.

Parameters also have local scope. Let's change the code a bit so we can see how this works:

#### CODE TO TYPE:

```
var message = "Loading...";

window.onload = function() {
  var message = "Done loading";
  var div = document.getElementById("container");
  div.innerHTML = message;
  updateMessage(message);
}

function updateMessage(msg) {
  console.log(message);
  console.log(msg);
  var div = document.getElementById("container");
  div.innerHTML = msg;
}
```



and **Preview** . In the console, you see two messages:

#### OBSERVE:

```
Loading...
Done loading
```

The first parameter is from the line `console.log(message)`. The value in the global variable `message` is displayed, not the value defined in the `window.onload` function, even though we're calling `updateMessage()` from that function. Also, we're passing the value of the local variable `message` from that function to `updateMessage()`, but giving it a new name as a parameter, `msg`. Remember that when you call a function and pass an argument, the parameter of the function gets a *copy* of the argument value, so `msg` gets a copy of the string "Done loading." That's the second message you see in the console, and it's also the message you see in the `<div>` in the web page.

`msg` is a local variable; it's local to the function `updateMessage()`. Try to display the value of `msg` in the console:

#### INTERACTIVE SESSION:

```
> msg
ReferenceError: msg is not defined
```

You can't access a local variable outside the function in which it is defined.

You might be curious about how variable shadowing works. And if there's a global object into which all the global variables are stashed, is there also a local object for local variables? We'll come back to these topics shortly, when we talk about **scope chains**.

## Hoisting

We've talked about the two kinds of scope that JavaScript has: global and local. Let's take a closer look at local scope, because sometimes local scope behaves in ways you might not expect, particularly if you have blocks of code in a function, and you're defining new variables within those blocks. By "block," we mean perhaps a loop or an if statement, where the "block" consists of all the statements inside the body of the loop or if statement (everything between the curly brackets {}):

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Hoisting </title>
  <meta charset="utf-8">
  <script>
    var icecream = ["vanilla", "chocolate", "strawberry"];
    function init() {
      var flavorButton = document.getElementById("getFlavorButton");
      flavorButton.onclick = checkFlavor;
    }
    function checkFlavor() {
      console.log(flavor);
      if (icecream.length > 0) {
        var div = document.getElementById("container");
        var flavor = document.getElementById("flavor").value;
        if (flavor) {
          for (var i = 0; i < icecream.length; i++) {
            if (icecream[i] == flavor) {
              var found = true;
              div.innerHTML = "We have " + flavor;
              break;
            }
          }
          if (!found) {
            div.innerHTML = "Sorry, we don't have " + flavor;
          }
        }
      }
      console.log(flavor);
    }
    window.onload = init;
  </script>
</head>
<body>
<div id="container">Enter a flavor of ice cream you'd like: </div>
<form>
  <input type="text" id="flavor">
  <input type="button" id="getFlavorButton" value="Check flavor">
</form>
</body>
</html>
```



Save this in your **/AdvJS** folder as **hoisting.html**, and **Preview**. You see a message asking you to enter an ice cream flavor, a form input to enter the flavor, and a button to submit the form.

Before you submit the form, take a close look at the code and make sure you understand how it works:

**OBSERVE:**

```
var icecream = ["vanilla", "chocolate", "strawberry"];
function init() {
  var flavorButton = document.getElementById("getFlavorButton");
  flavorButton.onclick = checkFlavor;
}
function checkFlavor() {
  console.log(flavor);
  if (icecream.length > 0) {
    var div = document.getElementById("container");
    var flavor = document.getElementById("flavor").value;
    if (flavor) {
      for (var i = 0; i < icecream.length; i++) {
        if (icecream[i] == flavor) {
          var found = true;
          div.innerHTML = "We have " + flavor;
          break;
        }
      }
      if (!found) {
        div.innerHTML = "Sorry, we don't have " + flavor;
      }
    }
  }
  console.log(flavor);
}
```

We set up a **click handler for the form button that will call the checkFlavor() function** when you click the button. In the **checkFlavor()** function we use a variable named **flavor** to hold the value you'll enter into the form. Look at where that variable is defined in the function; we display its value in the console twice: once at the top of the function, before the **flavor** variable is defined, and once at the end of the function, just before the function finishes, and outside the **if block** in which **flavor** is defined.

What do you think you'll see in the console when you run this code by entering an ice cream flavor and clicking the button?

Okay, make sure your console is open, and enter "vanilla." You see the message "We have vanilla" in the page. Check out the values displayed in the console.

**OBSERVE:**

```
undefined
vanilla
```

The first **console.log(flavor)** displays **undefined**, while the second **console.log(flavor)** displays **vanilla**.

You might have expected to get an error for the first **console.log(flavor)** call, because the **flavor** variable hasn't been defined yet, and usually when you try to access a variable that isn't defined, you get a Reference Error. (If you need to refresh your memory about Reference Errors, just go back to the console and enter the name of a variable that's not defined by this program, like x.)

Why don't we get a Reference Error when we try to access **flavor** before it's been defined, or after the block that encloses the definition of **flavor** has ended? The answer is: *hoisting*. Hoisting is an informal name for a quirky behavior in JavaScript: no matter where you define a variable within a function, the variable *declaration* is moved (or "hoisted") up to the top of the function. Read that again. Notice that *only* the declaration is moved; *not* the initialization. So, it's as if you'd written the **checkFlavor()** function like this:

## OBSERVE:

```
function checkFlavor() {
  var flavor;
  console.log(flavor);
  if (icecream.length > 0) {
    var div = document.getElementById("container");
    var flavor = document.getElementById("flavor").value;
    if (flavor) {
      for (var i = 0; i < icecream.length; i++) {
        if (icecream[i] == flavor) {
          var found = true;
          div.innerHTML = "We have " + flavor;
          break;
        }
      }
      if (!found) {
        div.innerHTML = "Sorry, we don't have " + flavor;
      }
    }
  }
  console.log(flavor);
}
```

`flavor` is declared when we access it in the first `console.log(flavor);`. It's declared, but it's not initialized, so the value of `flavor` is **undefined**. That's why you see **undefined** as the result of the first `console.log(flavor);`. Then, we set its value to the value you type into the form, and that remains the value when we reach the second `console.log(flavor);`.

Hoisting take place for *all* the local variables in `checkFlavor()`. Can you find the other variables in the function that are hoisted?

If you have experience with another language, perhaps Java or C#, this hoisting behavior may surprise you, because many languages have a third type of scope: block scope. In block scope, a variable like `flavor` would be defined *only* within the block where it's declared and initialized, and not outside that block. In that case, you *would* get a reference error if you tried to access `flavor` in the two `console.log(flavor);` statements. However, JavaScript does not have block scope. It only has global and local scope. All local variables are visible everywhere within a function, even if they are declared and initialized within a block.

**Note** As of this writing, JavaScript does not have block scope, but it will probably be added at some point.

## Nested Functions

Sometimes in JavaScript, we nest functions inside other functions. Usually these are "helper" functions that are used only by the function enclosing the nested functions. Let's take a look at an example, and then talk about how scope works for nested functions.



**CODE TO TYPE:**


```
<!doctype html>
<html>
<head>
  <title> Nested Functions </title>
  <meta charset="utf-8">
  <style>
    .red {
      background-color: red;
    }
    .blue {
      background-color: lightblue;
    }
    .pink {
      background-color: pink;
    }
  </style>
  <script>
    window.onload = function() {
      var theId = "list";

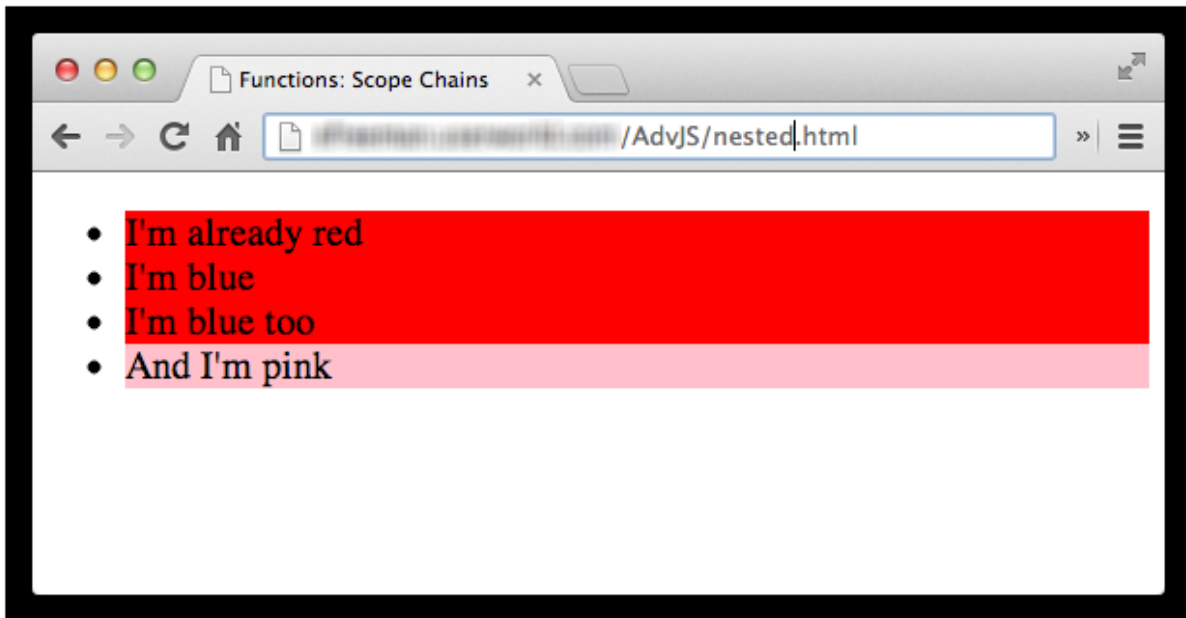
      findElement(theId);
    }

    function findElement(id) {
      var color = "red";
      var el = document.getElementById(id);
      if (el) {
        changeAllBlueChildren(el);
      }

      function changeAllBlueChildren(el) {
        for (var i = 0; i < el.childElementCount; i++) {
          var child = el.children[i];
          if (child.tagName.toLowerCase() == "li") {
            var theClass = child.getAttribute("class");
            if (theClass == "blue") {
              child.setAttribute("class", color);
            }
          }
        }
      }
    }
  </script>
</head>
<body>
  <ul id="list">
    <li class="red">I'm already red</li>
    <li class="blue">I'm blue</li>
    <li class="blue">I'm blue too</li>
    <li class="pink">And I'm pink</li>
  </ul>
</body>
</html>
```



Save this in your **/AdvJS** folder as **nested.html**, and  **Preview**. The first three items in the list are red, and the last one is pink. In the HTML and CSS, the second and third items are in the "blue" class, so normally the background of those items would be "lightblue," but we changed that using the code.



In the code, you can also see that we have a `window.onload` function that is called when the page is loaded; that function calls `findElement()`, which finds the element with the id "list," and calls `changeAllBlueChildren()`. `changeAllBlueChildren()` iterates through all of the child elements of the list (all the `<li>` elements), and if the class of the element is "blue," `changeAllBlueChildren()` changes the class to "red."

```
OBSERVE:

function findElement(id) {
  var color = "red";
  var el = document.getElementById(id);
  if (el) {
    changeAllBlueChildren(el);
  }

  function changeAllBlueChildren(el) {
    for (var i = 0; i < el.childElementCount; i++) {
      var child = el.children[i];
      if (child.tagName.toLowerCase() == "li") {
        var theClass = child.getAttribute("class");
        if (theClass == "blue") {
          child.setAttribute("class", color);
        }
      }
    }
  }
}
}
```

Here we're exploring nested functions. The `findElement()` function has a **nested function** in it: a function defined within a function. Unlike global functions like `findElement()` and `window.onload` function, `changeAllBlueChildren()` is a local function, visible only within `findElement()`. If you try to access the function in the console (or try to call it from outside of the `findElement()` function), you'll get a reference error:

```
INTERACTIVE SESSION:

> changeAllBlueChildren
ReferenceError: changeAllBlueChildren is not defined
```

Essentially, `changeAllBlueChildren()` is hidden except within `findElement()`. (We'll come back to the concept of data hiding later in the course).

We're using a function declaration to declare the `changeAllBlueChildren()` function. What do you think would happen if we changed the code to declare the function using a function expression instead?

CODE TO TYPE:

```
function changeAllBlueChildren(e1) {  
var changeAllBlueChildren = function(e1) {  
    ...  
}
```



and **Preview**

OBSERVE:

```
> Uncaught TypeError: undefined is not a function
```

Just like other local variables in a function, the `changeAllBlueChildren` variable is *hoisted*. So now, `changeAllBlueChildren` is implicitly declared at the top of the function but is undefined (just like all the other local variables). Since we don't assign the function expression to the variable until *after* we call the function, we get an error. We're trying to call the variable `changeAllBlueChildren` while it's still undefined, and before it is assigned the function expression.

Go ahead and change your code back to use a function declaration instead. The function declaration is also defined *after* we call the function, but, unlike a function defined with a function expression, a nested function declaration is visible throughout the function in which it's nested. This is similar to the way functions declared at the global level work: you can put them at the bottom of your code, but access them anywhere in your code. Here, we've placed the `changeAllBlueChildren` function declaration at the bottom of the `findElement()` function, but now (using a function declaration rather than a function expression) it's visible (and defined!) throughout the `findElement()` function.

## Lexical Scoping

Now let's take a look at what happens to the scope of variables when you have nested functions. Remember earlier we said local variables are visible throughout the function in which they are declared. Let's see where the variables in this program are visible.

OBSERVE:

```
window.onload = function() {  
    var theId = "list";  
    findElement(theId);  
}  
  
function findElement(id) {  
    var color = "red";  
    var e1 = document.getElementById(id);  
    if (e1) {  
        changeAllBlueChildren(e1);  
    }  
  
    function changeAllBlueChildren(e1) {  
        for (var i = 0; i < e1.childElementCount; i++) {  
            var child = e1.children[i];  
            if (child.tagName.toLowerCase() == "li") {  
                var theClass = child.getAttribute("class");  
                if (theClass == "blue") {  
                    child.setAttribute("class", color);  
                }  
            }  
        }  
    }  
}
```

We'll start with `theId`. This variable is defined in the `window.onload` function, and so is not visible globally or in `findElement()`. However, we pass it to `findElement()`, which gets a copy of its value, in the parameter

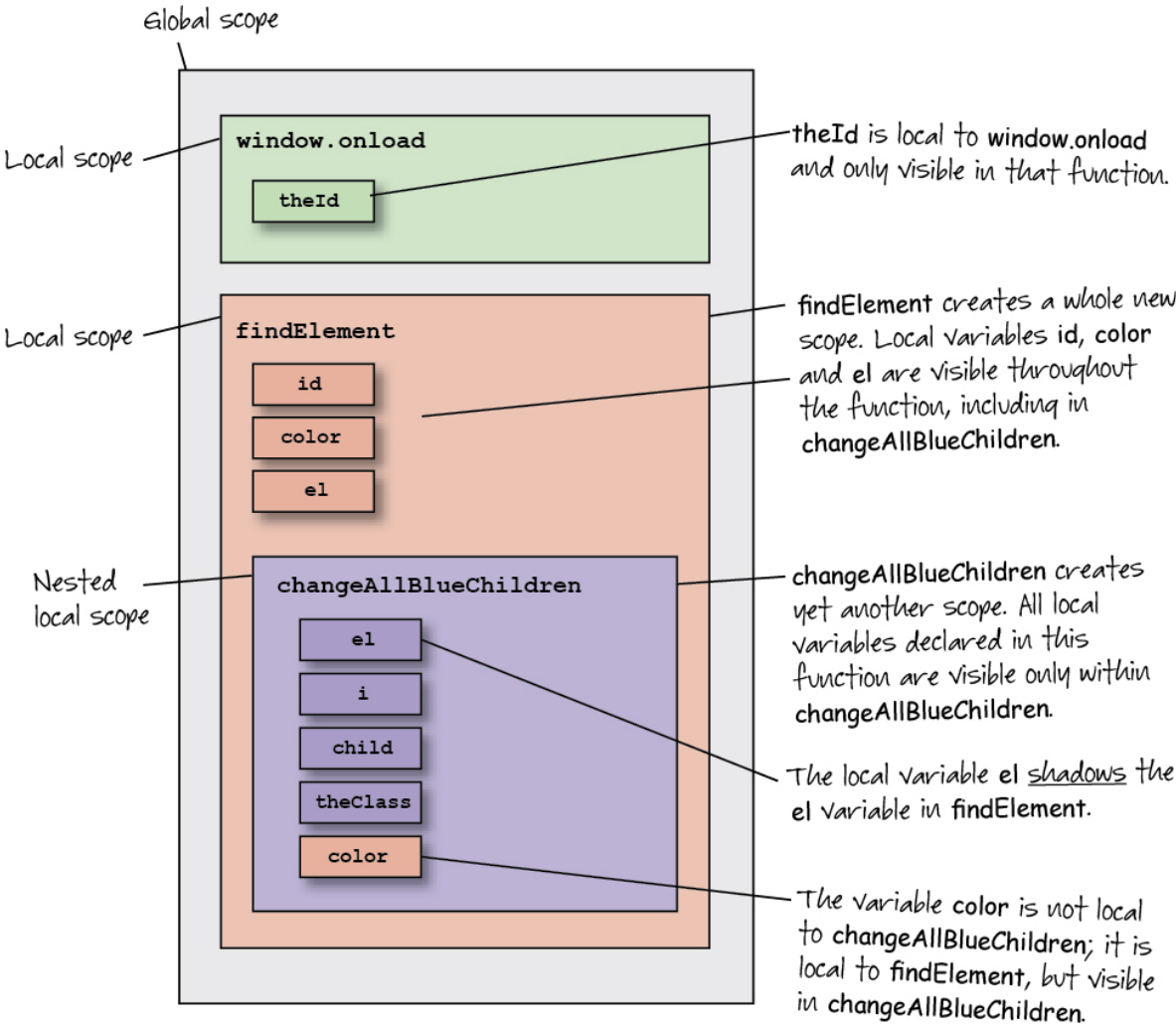
variable `id`. `id` is local to `findElement()`, so we can access it anywhere in that function, but again, not globally. Similarly, `color` and `el` are local variables, visible anywhere in `findElement()`.

Next, look at `changeAllBlueChildren()`. We have a parameter, `el`, and several local variables, including `i`, `child`, and `theClass`, all of which are local to `changeAllBlueChildren()`.

There's some interesting stuff happening here. We're *using* the variable `color` inside `changeAllBlueChildren()`, even though `color` is not defined in `changeAllBlueChildren()`, and it's not a global variable. This is possible because `changeAllBlueChildren()` is nested *within* `findElement()`, `color` is visible inside the nested function, so we can access it just as if it were a local (or global) variable inside `changeAllBlueChildren()`. This is known as *lexical scoping*. That means, when you are using a variable, like `color`, you figure out the value of the variable by first looking in the local scope (that is, in `changeAllBlueChildren()`), and then in the next outer scope. Typically, the next outer scope is the global scope, but in this case, because `changeAllBlueChildren()` is nested within another function, the next outer scope is `findElement()`. That's where `color` is defined, so that's the value we use in `changeAllBlueChildren()`.

Take a look at `el`. We have the parameter, `el`, which is local to `changeAllBlueChildren()`, and we have the variable `el` in `findElement()` which is visible throughout `findElement()`, including within `changeAllBlueChildren()`! So which value do we use? Well, remember that the `el` defined within `changeAllBlueChildren()` will *shadow* the `el` in `findElement()`. How? Because of lexical scoping. When we use `el` in `changeAllBlueChildren()`, we first look for it in the local scope, and we find its value there, so that's the one we use.

If you don't find a variable within a local scope at all, then you look in the global scope. If it's not there, you get a Reference Error.



Well, we've talked about how functions are good for organizing bits of code. That's what we're doing here. We put all the code related to changing the blue children of an element to red in this function, which makes the code easier to read and understand, and also creates a chunk of code that's easy to reuse. In this case, we only call the `changeAllBlueChildren()` once, but you can imagine that we might end up calling it multiple

times. By nesting the function inside **findElement()**, we keep it hidden from other code that doesn't need to know about it, and organize our code so that the related bits are together. If we wanted to use **changeAllBlueChildren()** somewhere else in our code, we'd have to move it out of **findElement()** so it would be accessible to other code to call.

There is a downside to nested functions: a nested function like **changeAllBlueChildren()** has to be recreated every time you call the function in which it's nested, while functions defined at the global level are created only once and stick around for the duration of your program. In our case, that's okay; we call **findElement()** just once, so we're creating **changeAllBlueChildren()** just once. In small to medium-sized programs, you might find that the organizational benefit of nested functions outweighs the performance hit. However, if you're building an application that needs to be as fast as possible, you'll want to avoid nested functions.

## Scope Chains

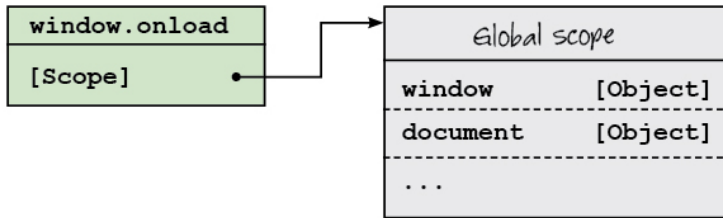
You know that JavaScript has two kinds of scope: global and local. You also know that scope works "lexically." When you are looking to *resolve* a variable (determine what its value is when you use it), first you look in the local scope of the function you're in, if you don't find it, you look in the surrounding scope, and so on, until you get to the global scope. If you don't find the variable in any of those places, you get an error.

The way scope works behind the scenes is through a *scope chain*. A scope chain is created in two stages: the first stage is when the function is defined, and the second, when the function executes. When you define a function, the initial scope chain is created. You can think of the scope chain as a list of pointers from the function to the scopes that surround the function at define time. These scopes are in order, so that the scope at the top of the chain (the first position) is the scope immediately surrounding the function (usually the global scope, unless the function is nested), and subsequent scopes are further out (like the layers in the previous diagram).

In the second stage, when the function executes, an activation object is created. This object represents the state of the function as it executes, so it contains all the local variables, as well as **this** (which is usually the global object, **window**). The activation object is added to the *top* (the first position) of the scope chain. When a function is executing and it comes across a variable and needs to resolve that variable to know what the value is, the function starts at the first position in the scope chain, which is the activation object. If the function finds the variable there (that is, the variable is a local variable to the function), it stops there, and uses that value. If the function doesn't find the variable there, it looks in the next position in the scope chain, and so on. The function continues down the chain until it finds a scope that contains the variable. If it gets to the end—the global scope is always the last stop in the chain—and hasn't found the variable, then we get a Reference Error.

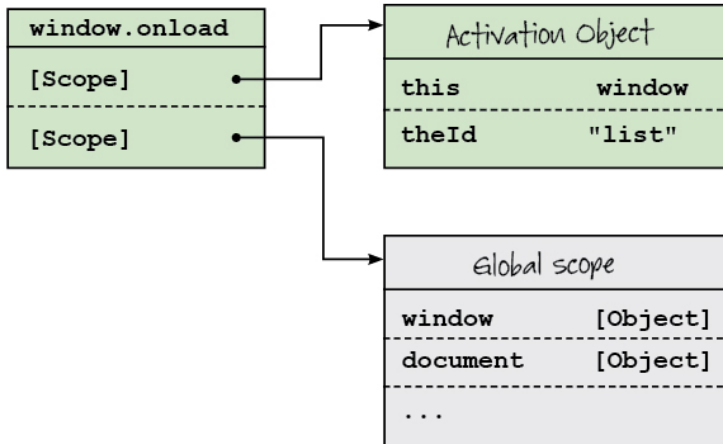
Let's take a look at the scope chains for the functions in our specific example. The **window.onload** function is defined at the top level, so its define-time scope chain is the global object. When **window.onload** is called (by the browser, when the page is loaded), an activation object is added to the top of the scope chain. So, when we look for the value of the variable **thead**, we find it in the activation object, and don't have to look any further down the chain.

When we define the `window.onload` function:



The scope chain is set up when we define a function.

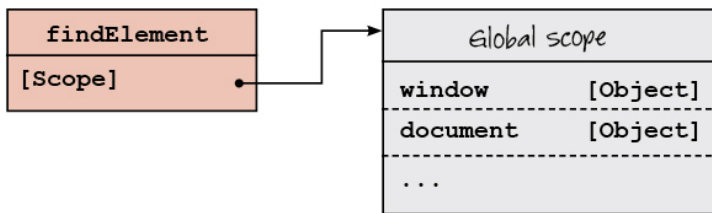
When we execute the `window.onload` function:



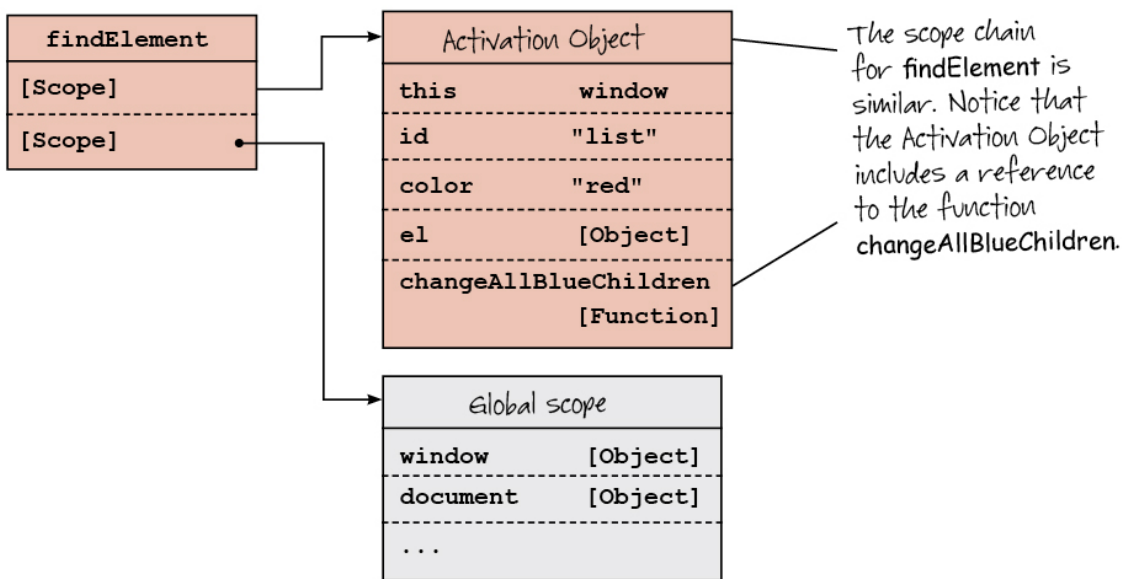
When we use a variable, like `theId`, in our code, we look at the first scope in the chain (the Activation Object), and if we don't find it there, we look at the second scope in the chain (the Global Scope). If we don't find it there, we get an error.

Similarly, `findElement()` is globally defined, so at define time, the global object is added to the scope chain, and at execution time—when we call the function—the activation object is added to the chain. That's where we look first to find the values of the variables used in the function, like `id` and `el`, as well as `document`. `document` is not defined in the activation object, so we look in the global object and find `document` there.

When we define the `findElement` function:

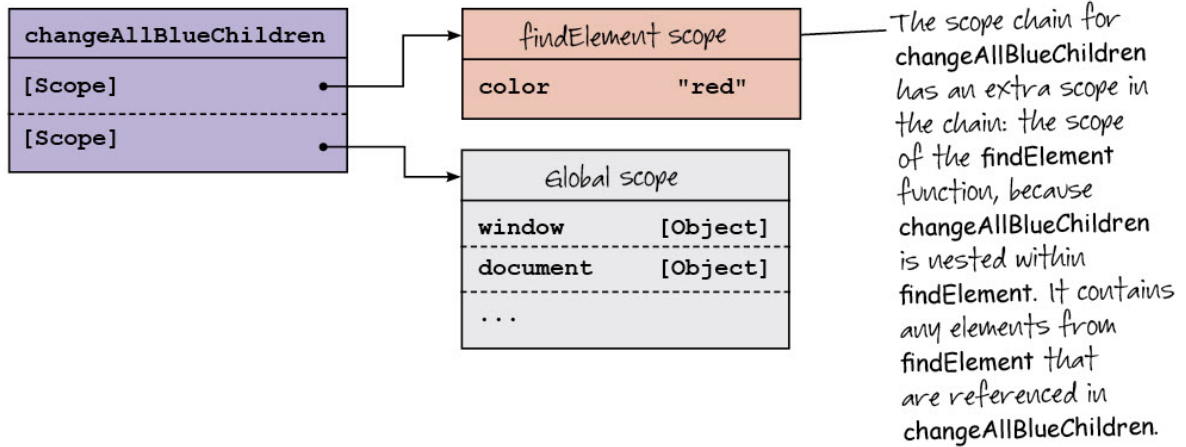


When we execute the `findElement` function:

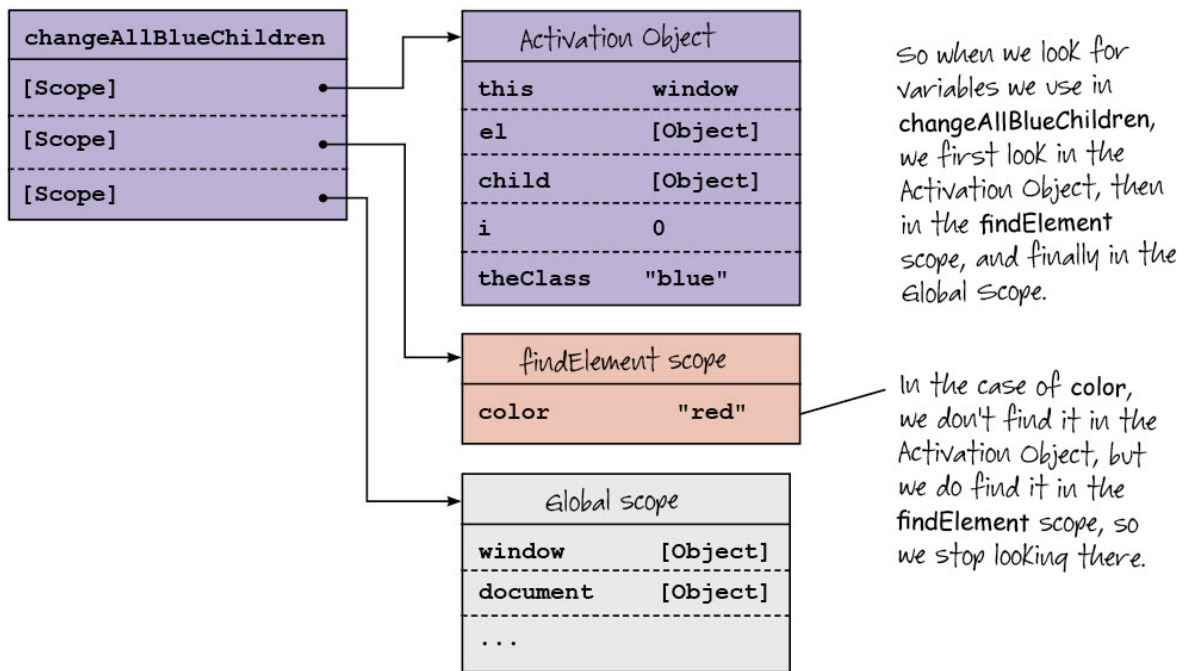


When we call `findElement()`, `changeAllBlueChildren()` is defined. At define time, we have an extra scope object in the chain: the `findElement()` scope object. Because `changeAllBlueChildren()` is defined within the `findElement()` function, it's defined in the `findElement()` function's scope. The extra scope object contains any variables that are referenced from the nested function. In this case, that's just the `color` variable. When we call `changeAllBlueChildren()`, its activation object is added to the chain. When we look for the variable `color`, first we look in the activation object, but since we don't find it there, we look in the `findElement()` scope, find it, and we stop looking.

When we define the `changeAllBlueChildren` function:



When we execute the `changeAllBlueChildren` function:

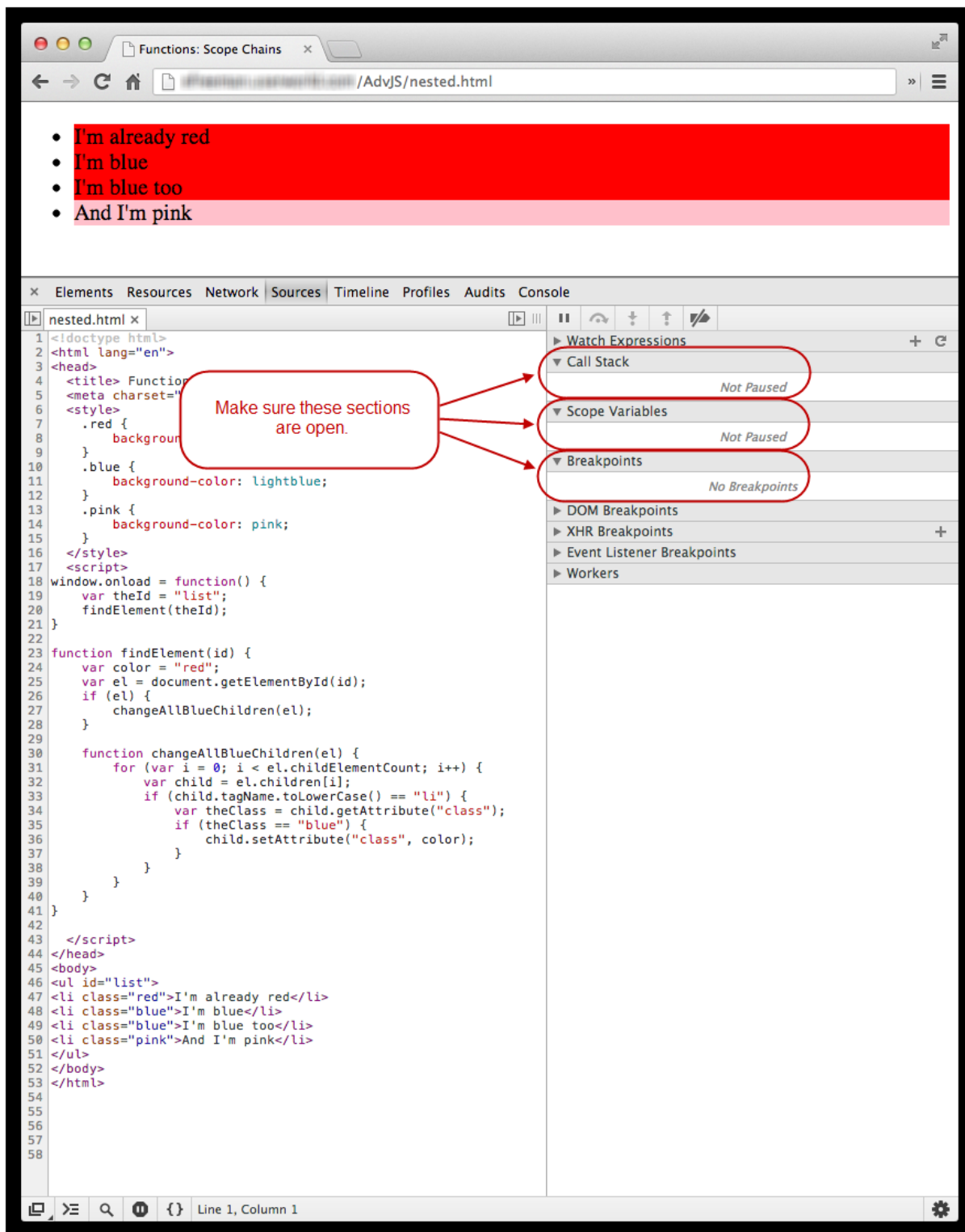


## Inspecting the Scope Chain

You can use the Chrome developer tools to see the scope chain in action as your code runs. Make sure you have the file `nested.html` loaded into a browser page and the console window open in the Chrome browser (unfortunately, the other browsers' built-in tools don't offer the same capability yet, so you'll need Chrome to follow along).

Click the **Sources** tab. If you don't see anything in the left panel in the console, click the small right-pointing **Show Navigator** arrow at the top left of the console and select your file, `nested.html`. Once you do that, you see the source code of the file:





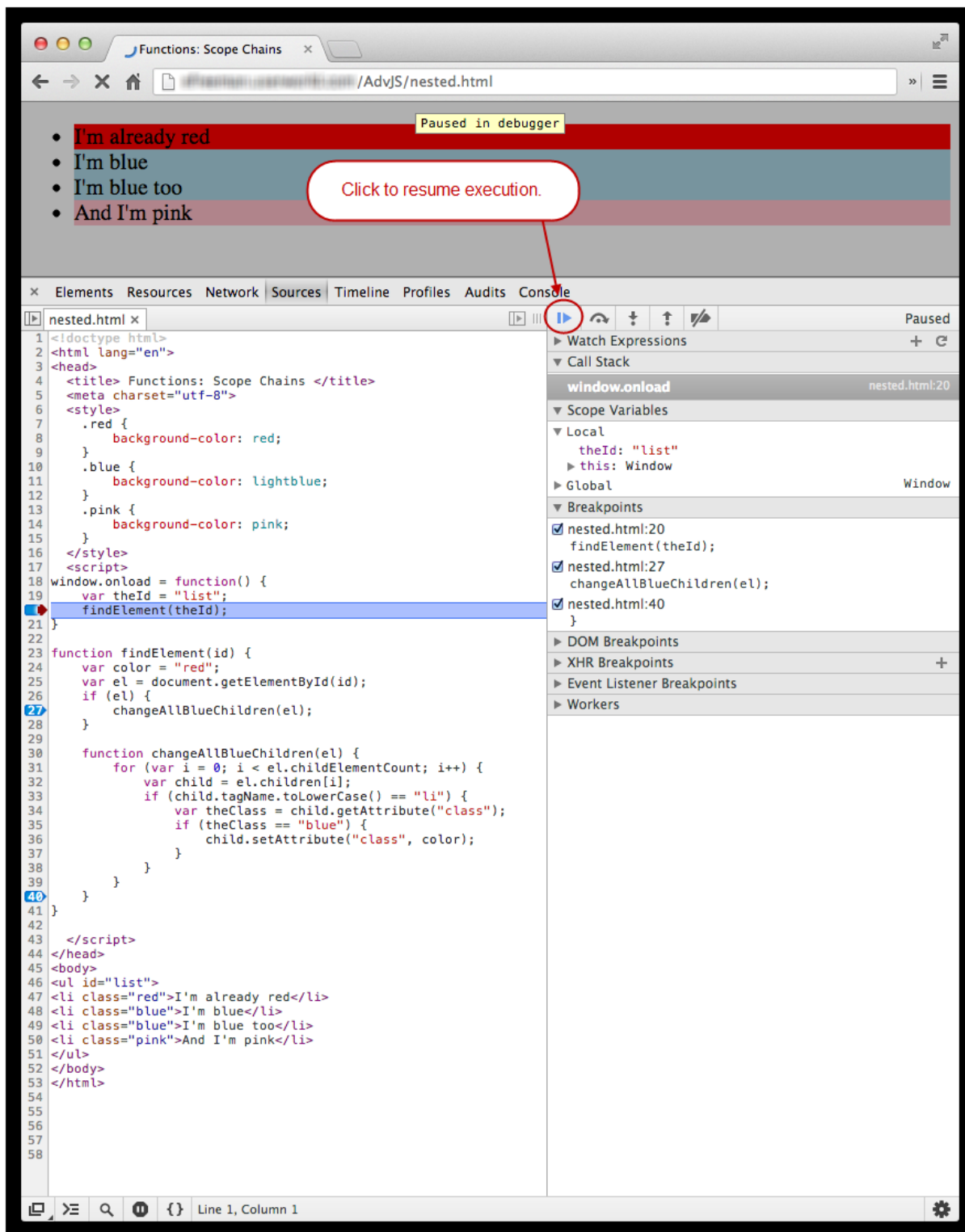
In the panel on the right side, make sure you have the Call Stack, Scope Variables, and Breakpoints sections open (the arrows next to them are pointed down).

Now, add breakpoints to your code. A breakpoint is a way to tell the browser to stop executing your code at a certain point. You add a breakpoint by clicking on one of the line numbers in the far left side of the left panel. When you add a breakpoint, you'll see a little blue marker indicating the line of code where the breakpoint is located. Add three breakpoints: one on the line where we call **findElement()** in **window.onload**; one on the line where we call **changeAllBlueChildren()** in **findElement()**, and one at the very bottom of **changeAllBlueChildren()** (the closing curly brackets for the function, which is the second to last curly bracket in the file). Look in the panel on the right, under Breakpoints, to verify that you've clicked on the correct lines.

The screenshot shows a web browser window with a page titled "Functions: Scope Chains". The page content is a list of four items: "I'm already red", "I'm blue", "I'm blue too", and "And I'm pink". The list is styled with different background colors: red for the first item, light blue for the second, blue for the third, and pink for the fourth. Below the page content is the Chrome DevTools interface. The Sources panel shows the HTML and JavaScript code for the page. Three breakpoints are set in the JavaScript code: one at line 20 (the start of the `findElement` function), one at line 27 (the start of the `changeAllBlueChildren` function), and one at line 40 (the end of the `changeAllBlueChildren` function). The Breakpoints panel on the right shows these three breakpoints are active. The status bar at the bottom indicates "Line 1, Column 1".

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4   <title> Functions: Scope Chains </title>
5   <meta charset="utf-8">
6   <style>
7     .red {
8       background-color: red;
9     }
10    .blue {
11      background-color: lightblue;
12    }
13    .pink {
14      background-color: pink;
15    }
16  </style>
17  <script>
18  window.onload = function() {
19    var theId = "list";
20    findElement(theId);
21  }
22
23  function findElement(id) {
24    var color = "red";
25    var el = document.getElementById(id);
26    if (el) {
27      changeAllBlueChildren(el);
28    }
29
30    function changeAllBlueChildren(el) {
31      for (var i = 0; i < el.childElementCount; i++) {
32        var child = el.children[i];
33        if (child.tagName.toLowerCase() == "li") {
34          var theClass = child.getAttribute("class");
35          if (theClass == "blue") {
36            child.setAttribute("class", color);
37          }
38        }
39      }
40    }
41  }
42
43  </script>
44  </head>
45  <body>
46  <ul id="list">
47  <li class="red">I'm already red</li>
48  <li class="blue">I'm blue</li>
49  <li class="blue">I'm blue too</li>
50  <li class="pink">And I'm pink</li>
51  </ul>
52  </body>
53  </html>
54
55
56
57
58
```

Now, reload the page. Don't worry, your breakpoints will stay in place. When you do, you see a "Paused in debugger" message in the web page at the top, and the line of code at the first breakpoint is highlighted which indicates that the execution has paused at that line:



In the Call Stack section in the panel on the right, you see **window.onload**. That's because we called **window.onload**, and we paused execution just before we call **findElement()**. Now, look at the Scope Variables section. This is the scope chain. At the top of the chain is the local scope; you can see the local variables defined there, including **theId** and **this**. Next you see the global scope, with all of its usual content (you can open it up to see, but it contains many properties, so be sure and close it when you're finished).

Now, we want to continue the execution of the code until we hit the next breakpoint. Click the small button that looks like a "play" button (**Resume script execution**) at the top left of the right panel. When you click this button, execution resumes, until it hits the next breakpoint located at the line where we call **changeAllBlueChildren()**:

Paused in debugger

- I'm already red
- I'm blue
- I'm blue too
- And I'm pink

Elements Resources Network Sources Timeline Profiles Audits Console

nested.html x

```

1 <!doctype html>
2 <html lang="en">
3 <head>
4   <title> Functions: Scope Chains </title>
5   <meta charset="utf-8">
6   <style>
7     .red {
8       background-color: red;
9     }
10    .blue {
11      background-color: lightblue;
12    }
13    .pink {
14      background-color: pink;
15    }
16  </style>
17  <script>
18  window.onload = function() {
19    var theId = "list";
20    findElement(theId);
21  }
22
23  function findElement(id) {
24    var color = "red";
25    var el = document.getElementById(id);
26    if (el) {
27      changeAllBlueChildren(el);
28    }
29
30    function changeAllBlueChildren(el) {
31      for (var i = 0; i < el.childElementCount; i++) {
32        var child = el.children[i];
33        if (child.tagName.toLowerCase() == "li") {
34          var theClass = child.getAttribute("class");
35          if (theClass == "blue") {
36            child.setAttribute("class", color);
37          }
38        }
39      }
40    }
41  }
42
43  </script>
44 </head>
45 <body>
46 <ul id="list">
47 <li class="red">I'm already red</li>
48 <li class="blue">I'm blue</li>
49 <li class="blue">I'm blue too</li>
50 <li class="pink">And I'm pink</li>
51 </ul>
52 </body>
53 </html>
54
55
56
57
58

```

Paused

Watch Expressions +

Call Stack

- findElement nested.html:27
- window.onload nested.html:20

Paused on a JavaScript breakpoint.

Scope Variables

Local

- changeAllBlueChildren: function changeAllBlueCh...
- color: "red"
- el: ul#list
- id: "list"
- this: Window

Global Window

Breakpoints

- nested.html:20 findElement(theId);
- nested.html:27 changeAllBlueChildren(el);
- nested.html:40 }

DOM Breakpoints

XHR Breakpoints +

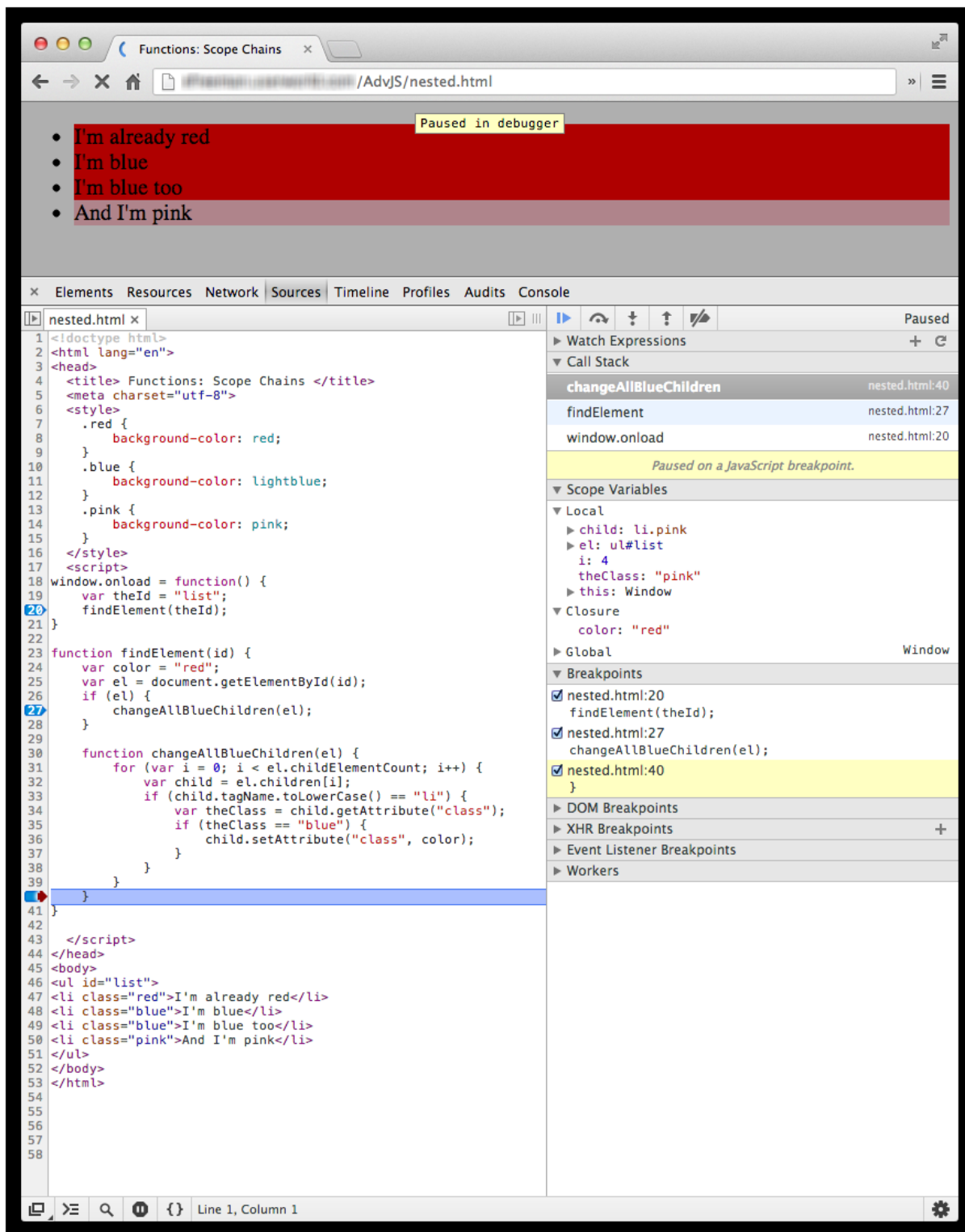
Event Listener Breakpoints

Workers

Line 1, Column 1

Again the execution stops just before we call the function, so we can inspect the call stack and the scope variables. Notice under Call Stack in the panel on the right, we now have **findElement()** on top, and **window.onload** below. That means that we called **findElement()** from **window.onload**. Now look at the Scope Variables. The top part of the scope chain includes the local variables defined by **findElement()**. Below that is the global scope. Before you move on, notice that the second and third items in the list are still blue because we haven't called **changeAllBlueChildren()** yet!

Continue the code execution by clicking the **Resume script execution** button. Now the execution stops on the very last line of the **changeAllBlueChildren()** function—on the curly brace, at the moment just before this function returns.



Now the second and third items in the list are red. Notice the Call Stack; you can see that **changeAllBlueChildren()** is at the top. We called **changeAllBlueChildren()** from **findElement()**, which we called from **window.onload** (the Call Stack is useful for tracking which functions are calling which!).

Look at the Scope Variables. You see that the local variables for the **changeAllBlueChildren()** function at the top of the scope chain. Below that is something called **Closure**. This is the scope chain object for the **findElement()** scope, containing the **color** variable. Twirl down (click so it's pointing downward) the arrow next to **Closure** so you can see it. (We'll explain why it's called Closure in another lesson). Then, at the bottom of the scope chain is the global scope.

Click **Resume script execution** one more time to continue execution and complete the code execution. The page returns to normal. Your breakpoints are still there (they will stay there until you delete them) so if you want to do this again, you can simply reload the page and go through the same steps. To remove the

breakpoints, click the blue markers and they will disappear. Once you've removed the breakpoints, if you reload the page, the page will behave as it usually does, and execute the code all the way through with no stops.

In this lesson, you learned all about scope, including the details of how it works with the scope chain. Nested functions, sometimes called "inner functions," create additional levels in the scope chain, so we explored how to create nested functions and how scope works when you have a nested function. Lexical scoping is a rule that lets us find the correct value of a variable to use by looking in each level of the scope chain, from the top (first) position, to the bottom position (always the global scope).

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Invoking Functions

## Lesson Objectives

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

- call a function as a function.
- call a function as a method.
- call a function as a constructor.
- call a function with `apply()` and `call()`.
- use **this** when we call functions in different ways.
- compare how **this** is defined in nested functions with how it is defined in global functions.
- control how **this** is defined with `apply()` and `call()`.
- use the **arguments** object to create functions that support a variable number of arguments.

## Invoking Functions

In JavaScript we call, or *invoke*, functions in many different ways. We call built-in functions, we create and call our own functions, we use functions as constructors, we call methods in objects, and more. In this lesson, we'll look at all the different ways we can call functions, and the reasons we use each.

**Note** *Invoke* is just another word for *call*. We'll use both terms interchangeably throughout the lesson.

## Different Ways to Invoke Functions

By now, you are familiar with how to call functions in JavaScript. It's difficult to write a script without calling at least one function. Perhaps you're calling a built-in function, like **alert**, or a function you've written yourself, or one that's supplied by a JavaScript library (like jQuery). Let's look at an example that has a couple of different kinds of function calls:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Calling a function </title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-latest.min.js"></script>
  <script>
    function callMeMaybe(number) {
      return "Call me at " + number;
    }

    window.onload = function() {
      var number = callMeMaybe("555-1212");
      $("body").append("<div>" + number + "</div>");
      alert("Content added");
    }
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **functionCalls.html**, and . You see the text "Call me at 555-1212" in the page, and an alert with the text, "Content added."

Here we see examples of two of the four different ways you can invoke functions in JavaScript:

- **as a function**
- **as a method**
- as a constructor (we'll get to this later)
- with `apply()` or `call()` (we'll get to this later too)

First, we create and invoke our own function, `callMeMaybe()`, as a function (that is, `callMeMaybe()` is a regular function that we call in the normal way we call functions). We also call a function supplied by jQuery, `$("#body").append()` (which is shorthand for the `jQuery()` function—and yes, `$` is a valid name for a function). Again, we call the `$("#body").append()` function as a function.

The second way to invoke a function is as a method. There are two method calls in this example: `$("#body").append()` and `alert()`. `$("#body").append()` is a method of the jQuery object returned from the `jQuery()` function. We use "dot notation" to call it, as we would any method.

So what about `alert()`? Well, remember earlier in the course, we said that built-in JavaScript functions like `alert()` are actually methods of the global `window` object, which is the default object used to run JavaScript programs in the browser. Because it's the default object, if you don't specify it for methods and properties (like with our call to `alert()`), JavaScript assumes you mean `window.alert()`. So calling `alert()` is actually a method call, not a function call.

There's actually a third method call *implied* in this code. Can you find it? It's the function assigned to the `window.onload` property. We don't call this method ourselves; the browser is calls it for us when the page has completed loading.

An important distinction between calling a function as a function and as a method is that (most of the time) you need to use the dot notation to call a method. You specify the name of the object, like `window`, and the name of the method, like `alert()`, and separate the two with a dot (period) to get `window.alert()`. To call a function, you use the function name.

Let's look at the third way to call a function:




**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Calling a function as a constructor </title>
  <meta charset="utf-8">
  <style>
    .square {
      background-color: lightblue;
      cursor: pointer;
    }
    .circle {
      background-color: orange;
      cursor: pointer;
    }
    .square p, .circle p {
      padding-top: 35%;
      text-align: center;
      -webkit-user-select: none;
      -moz-user-select: none;
      -ms-user-select: none;
      user-select: none;
    }
  </style>
  <script>
    function Square(id, name, size) {
      this.id = id;
      this.name = name;
      this.size = size;

      this.display = function() {
        var el = document.getElementById(this.id);
        el.style.width = this.size + "px";
        el.style.height = this.size + "px";
        el.innerHTML = "<p>" + this.name + "</p>";
        console.log(this.name + " has size " + this.size +
          ", and is a " + this.constructor.name);
      };
    }
    window.onload = function() {
      var square = new Square("s1", "square one", 100);
      square.display();
    }
  </script>
</head>
<body>
<div id="s1" class="square"></div>
</body>
</html>
```



Save this in your **/AdvJS** folder as **constructorCall.html**, and **Preview** . A light blue square with the text "square one" appears in the page. In the console, you see the message "square one has size 100, and is a Square."

As we mentioned in the lesson on constructing objects, when you invoke a function with the **new** keyword, you are calling that function as a constructor rather than as a function. So here, when we invoke the **Square()** function with **new Square(...)**, we're invoking that function as a constructor.

Constructors create an object, which we can reference with the **this** keyword to assign it property values, and that object is returned automatically by the constructor. Any function can be invoked as a constructor (although if you invoke a function not designed specifically as a constructor with **new**, you probably won't get a useful object back). As convention dictates, we begin constructor functions' names with an uppercase letter to distinguish these functions from regular functions.

Many objects have methods as properties. We create the Square object with the **Square()** constructor, so you can see here an example of invoking a function as a method, too: **square.display()**.

The fourth, and final, way to invoke a JavaScript function is indirectly, using the function object's **call()** and **apply()** methods. You haven't seen these methods yet, so we'll spend a significant amount of time on the subject here. The main reason to use these methods is to control how **this** is defined, so first we'll talk about what happens to **this** in each of the ways we invoke functions.

## What Happens to this When You Invoke a Function

The rules for how **this** is defined when you invoke a function depend on how you invoke the function and the context in which you invoke it. Let's take a look at some examples. We'll modify the previous example of the Square constructor to illustrate. Copy the previous file to a new file, **this.html**, and modify it as shown:



## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> What happens to this? </title>
  <meta charset="utf-8">
  <style>
    .square {
      background-color: lightblue;
      cursor: pointer;
    }
    .circle {
      background-color: orange;
      cursor: pointer;
    }
    .square p, .circle p {
      padding-top: 35%;
      text-align: center;
      -webkit-user-select: none;
      -moz-user-select: none;
      -ms-user-select: none;
      user-select: none;
    }
  </style>
  <script>
    var n = 0;

    function Square(id, name, size) {
      console.log("This at the top of the Square constructor: ");
      console.log(this);
      this.id = id;
      this.name = name;
      this.size = size;

      this.display = function() {
        console.log("This in the Square's display method: ");
        console.log(this);
        var el = document.getElementById(this.id);
        el.style.width = this.size + "px";
        el.style.height = this.size + "px";
        el.innerHTML = "<p>" + this.name + "</p>";
        console.log(this.name + " has size " + this.size +
          ", and is a " + this.constructor.name);
      };
      console.log("This at the bottom of the Square constructor: ");
      console.log(this);
    }
    window.onload = function() {
      console.log("This in window.onload: ");
      console.log(this);
      var square = new Square("s1", "square one", 100);
      setupClickHandler(square);
      square.display();
    }
    function setupClickHandler(shape) {
      console.log("This in setupClickHandler: ");
      console.log(this);
      var elDiv = document.getElementById(shape.id);
      elDiv.onclick = function() {
        console.log("This in click handler: ");
        console.log(this);
        n++;
        var counter = document.getElementById("counter_" + shape.id);
        counter.innerHTML = "You've clicked " + n + " times.";
      };
    }
  </script>
```

```
</head>
<body>
  <div id="s1" class="square"></div>
  <p id="counter_s1"></p>
</body>
</html>
```

 Save the file in your **/AdvJS** folder as **this.html** and  **Preview**. We added some calls to **console.log()** in several places (make sure you find them all now, because we'll refer to each of them below) We also added a whole new function, **setupClickHandler()**, which adds a click handler to the "s1" <div>, so that when you click on the "s1" <div> you'll see a message indicating how many times you've clicked on that <div>. Open the console (reload the page if you don't see any output), and you see messages like this:

```
OBSERVE:
> This in window.onload:
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
> This at the top of the Square constructor:
Square {}
> This at the bottom of the Square constructor:
Square {id: "s1", name: "square one", size: 100, display: function}
> This in setupClickHandler:
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
> This in the Square's display method:
Square {id: "s1", name: "square one", size: 100, numClicks: 0, display: function}
> square one has size 100, and is a Square
```

Each of these **console.log()** messages displays the value of **this** at a particular spot in your code.

**The first message** is from the **console.log()** in the **window.onload** function. When you call the method of an object, typically, **this** refers to the object. That's the case here: we called the function assigned to the **window.onload** property (and thus, that function is a method), so **this** in that method is the **window** object.

**The second message** is from the **console.log()** at the top of the **Square()** constructor function. Here, **this** is set to a brand new object (the one created by the constructor), and because we have yet to set any of its properties, that object is empty.

At the end of the **Square()** constructor, we see **the third message** after we add properties and methods to the object being created by the constructor. In the message we see a fully constructed **Square** object that will be returned to the code that called **new Square()**.

**The fourth message** is from the **console.log()** in the **setupClickHandler()**, which we call just before we call **square.display()**. The value of **this** in **setupClickHandler()** is also the **window** object, but for an entirely different reason. Here, **this** is set to the **window** object because **setupClickHandler()** is a globally defined function we've created, and the context in which it is being called is the **window** object—that is, the global object. Whenever you call a globally defined function as a function, **this** is set to the **window** object (unless you've set it to something else).

**The last two messages** are from the **console.log()**s in the **square.display()** method. **this** is set to the **Square** object; that is the method we called from **Square** object.

We don't see the message in the "s1" <div> click handler yet (assuming you haven't clicked on that <div> yet; if you have, just reload the page and don't click on it so that your messages match ours).

Now, click on the "s1" <div> (by clicking anywhere on the text). More text appears in the page, "You've clicked 1 times." and in the console, you see:

```
OBSERVE:
>This in click handler:
<div id="s1" class="square" style="width: 100px; height: 100px;">...</div>
```

This is the message from the `console.log()` in the function we've assigned to the `onclick` property of the "s1" `<div>`. Because we're assigning a function to the `click` property of an object (an element object representing the `<div>`), this function is actually a method. Within that method, `this` is set to the object that contains the method we're calling—that is, the "s1" `<div>`.

Try clicking again. Notice that we increment the global variable `n` each time you click in order to keep track of the total number of times you've clicked. (As an exercise, think about why `n` has to be a global variable here.)

So far so good; everything is pretty much as we'd expect. Now let's make a change and see what happens:

#### CODE TO TYPE:

```
...
var n = 0;

function Square(id, name, size) {
  console.log("This at the top of the Square constructor: ");
  console.log(this);
  this.id = id;
  this.name = name;
  this.size = size;
  this.numClicks = 0;

  this.display = function() {
    console.log("This in the Square's display method: ");
    console.log(this);
    var el = document.getElementById(this.id);
    el.style.width = this.size + "px";
    el.style.height = this.size + "px";
    el.innerHTML = "<p>" + this.name + "</p>";
    console.log(this.name + " has size " + this.size +
      ", and is a " + this.constructor.name);
  };

  this.click = function() {
    console.log("This in the Square's click method: ");
    console.log(this);
    this.numClicks++;
    document.getElementById("counter_" + this.id).innerHTML =
      "You've clicked " + this.numClicks + " times on " + this.name;
  };
  console.log("This at the bottom of the Square constructor: ");
  console.log(this);
}
window.onload = function() {
  console.log("This in window.onload: ");
  console.log(this);
  var square = new Square("s1", "square one", 100);
  setupClickHandler(square);
  square.display();
}
function setupClickHandler(shape) {
  console.log("This in setupClickHandler: ");
  console.log(this);
  var elDiv = document.getElementById(shape.id);
  elDiv.onclick = function() {
    console.log("This in click handler: ");
    console.log(this);
    n++;
    var counter = document.getElementById("counter_" + shape.id);
    counter.innerHTML = "You've clicked " + n + " times.";
  };
  elDiv.onclick = shape.click;
}
...
```



and **Preview** preview.

We replaced the click handler for the "s1" <div> that we defined in **setupClickHandler()** with a method of the **square** object, **square.click()** (we assign it to the **onclick** property with **shape.click** in the **setupClickHandler()** function since we're passing the square into a parameter named **shape**). Essentially the method performs the same task as the previous click handler did, except that now we keep track of the number of clicks on the book with a property of the **square** object, **square.numClicks**, rather than with a global variable.

Now try clicking on the "s1" <div> as you did before. It doesn't work! You see the text "You've clicked NaN times on the book" in the page. Recall that **NaN** means "Not a Number," so something went wrong with the counter. In addition, we don't see the name of the book like we should.

Look at the code for the **click()** method:

<b>OBSERVE:</b>
<pre> this.click = function() {   console.log("This in the Square's click method: ");   console.log(this);   this.numClicks++;   document.getElementById("counter_" + this.id).innerHTML =     "You've clicked " + <b>this.numClicks</b> + " times on " + <b>this.name</b>; }; </pre>

We don't see the correct values for either **this.numClicks** or **this.name**. Looking at the console again, you'll see that the value of **this** in the **click()** method is the "s1" <div> object (the element object), and *not* the **square** object:

<b>INTERACTIVE SESSION:</b>
<pre> &gt; This in the Square's click method: &lt;div id="s1" class="square" style="width: 100px; height: 100px;"&gt;...&lt;/div&gt; </pre>

So here, the "s1" <div> object replaces the normal value of **this** in the method **square.click()**. In fact, **this** will be defined as the target element object in any function that you use as the handler for an event like "click." This behavior comes in handy when we want to access the element that was clicked in the click handler, but in this case, the behavior is trading on our expectations of what **this** should be in the method of the **square** object.

Fortunately, there's an easy way around it. Instead of writing:

<b>OBSERVE:</b>
<pre> elDiv.onclick = shape.click; </pre>

we can write:

<b>OBSERVE:</b>
<pre> elDiv.onclick = function() {   shape.click(); } </pre>

Now, **this** is set to the "s1" <div> in the outer click handler function (the anonymous function we're assigning to the **onclick** property of the **elDiv**), but when we call **shape.click()** from within that function, **this** gets set to the **square** object that we passed into **setupClickHandler()** (because now we're calling the method normally, as a method of an object, rather than as an event handler). If you need the "s1" <div> object in the click handler for some reason, you could pass **this** to the method:

#### OBSERVE:

```
elDiv.onclick = function() {
    shape.click(this);
}
```

...and change the **click()** method in the **Square** constructor to have a parameter to take this argument.

We don't need the "s1" <div> in the **click()** method, so for now, just change your code like this:

#### CODE TO TYPE:

```
...
function setupClickHandler(book) {
    var elDiv = document.getElementById(shape.id);
elDiv.onclick = shape.click;
    elDiv.onclick = function() {
        console.log("This in click handler: ");
        console.log(this);
        shape.click();
    };
}
...
```



and **Preview**. Now the page updates correctly when you click the "s1" <div>, and in the console, you'll see that **this** is set to the "s1" <div> object in the click handler function, and to the **square** object in the **square.click()** method, as you'd expect.

#### Note

JavaScript's treatment of **this** in event handlers is a tricky subject. If you use inline event handlers (not recommended!), such as **<p onclick="click()">**, **this** will refer to the global window object, not the <p> element object. Similarly, if you use the (now old) **attachEvent()** function in IE8 and earlier, **this** will refer to the window object. In order to maintain consistency when handling **this**, we recommend that you use the **onclick** property to set your click handlers (and other common event handlers), or **addEventListener()** if you know all your users are on IE9+. In both of these cases, **this** will be set to the element that is clicked in the event-handling function.

The key point to understand in this section of the lesson is that the value of **this** is different depending on how you invoke a function, and you'll want to know what the value of **this** will be in these different situations.

## Nested Functions

The value of **this** in nested functions might not be what you expect. Let's check out an example. Create a new HTML file as shown:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Nested functions </title>
  <meta charset="utf-8">
  <script>
function outer() {
  console.log("outer: ", this);
  inner();

  function inner() {
    console.log("inner: ", this);
  }
}
outer();
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **self.html**, and **Preview** . In the console, you see:

#### OBSERVE:

```
outer:
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
inner:
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
```

The first result should not be a surprise: we know that when you call a globally defined function, **this** is set to the global window object.

The value of **this** is the window object in the **inner()** function too. You might be surprised if you expected **this** to be defined as the function **outer** (since **outer** is an object—yes, functions are objects too!). Another thing to remember about **this**.

How about when you're inside an object constructor and call a nested function?



#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Nested functions </title>
  <meta charset="utf-8">
  <script>
function outer() {
  console.log("outer: ", this);
  inner();

  function inner() {
    console.log("inner: ", this);
  }
}
outer();

function MakeObject() {
  this.aProperty = 3;

  console.log("outer object: ", this);

  function inner() {
    console.log("inner to object: ", this);
  }
  inner();
}
var outerObject = new MakeObject();
</script>
</head>
<body>
</body>
</html>
```



and **Preview**. The results you see this time might be even more surprising:

#### OBSERVE:

```
outer object: MakeObject {aProperty: 3}
inner to object:
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
```

**this** is defined to be the object we're creating in the **MakeObject()** constructor in the first message (as expected), but inside of the **inner()** function (that defines and calls inside the constructor). **this** is defined as the global window object. We "lose" the value of **this** (that is, the object we're creating) in the nested function.

As a result of this behavior, a common idiom in JavaScript is to "save" the value of **this** in another variable so it's accessible to the nested function, like this:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Nested functions </title>
  <meta charset="utf-8">
  <script>
function outer() {
  console.log("outer: ", this);
  inner();

  function inner() {
    console.log("inner: ", this);
  }
}
outer();

function MakeObject() {
  this.aProperty = 3;

  console.log("outer object: ", this);

  var self = this;
  function inner() {
    console.log("inner to object: ", thisself);
  }
  inner();
}
var outerObject = new MakeObject();
</script>
</head>
<body>
</body>
</html>
```



and **Preview**  preview. Now you see:

#### OBSERVE:

```
outer object:  MakeObject {aProperty: 3}
inner to object:  MakeObject {aProperty: 3}
```

We saved the value of **this** in the variable **self** before calling **inner()**. Because of lexical scoping, the **inner()** function can see the value of **self**, and so can access the object being created by the constructor.

#### Note

The behavior we've just described (**this** defined as the global window object in nested functions) may change in a later version of JavaScript.

## When You Want to Control How **this** is Defined

So you've seen how **this** is defined when you invoke (global) functions as functions, when you invoke functions as constructors, when you invoke functions as methods, and in the special cases when you invoke functions as click event handlers, and when you invoke nested functions.

In all these cases, JavaScript rules determine how **this** is defined. But what if you want to take control and define your own value for **this** when you invoke a function?

That's when we use **apply()** and **call()**. These methods are designed specifically to allow you to define the value of **this** inside a function you invoke with **apply()** or **call()**.

Let's say you decide to add another shape to your program. Edit **this.html** as shown:

**CODE TO EDIT: this.html**

```
...
<script>
  function Square(id, name, size) {
    this.id = id;
    this.name = name;
    this.size = size;
    this.numClicks = 0;

    this.display = function() {
      var el = document.getElementById(this.id);
      el.style.width = this.size + "px";
      el.style.height = this.size + "px";
      el.innerHTML = "<p>" + this.name + "</p>";
      console.log(this.name + " has size " + this.size +
        ", and is a " + this.constructor.name);
    };

    this.click = function() {
      console.log("This in the Square's click method: ");
      console.log(this);
      this.numClicks++;
      document.getElementById("counter_" + this.id).innerHTML =
        "You've clicked " + this.numClicks + " times on " + this.name;
    };
  }

  function Circle(id, name, radius) {
    this.id = id;
    this.name = name;
    this.radius = radius;
    this.numClicks = 0;

    this.display = function() {
      var el = document.getElementById(this.id);
      el.style.width = (this.radius * 2) + "px";
      el.style.height = (this.radius * 2) + "px";
      el.style.borderRadius = this.radius + "px";
      el.innerHTML = "<p>" + this.name + "</p>";
      console.log(this.name + " has radius " + this.radius +
        ", and is a " + this.constructor.name);
    };
    this.click = function() {
      this.numClicks++;
      document.getElementById("counter_" + this.id).innerHTML =
        "You've clicked " + this.numClicks + " times on " + this.name;
    };
  }


  window.onload = function() {
    var square = new Square("s1", "square one", 100);
    setupClickHandler(square);
    square.display();

    var circle = new Circle("c1", "circle one", 50);
    setupClickHandler(circle);
    circle.display();
  }

  function setupClickHandler(shape) {
    var elDiv = document.getElementById(shape.id);
    elDiv.onclick = function() {
      shape.click();
    }
  }
</script>
</head>
```

```
<body>
  <div id="s1" class="square"></div>
  <div id="c1" class="circle"></div>
  <p id="counter_s1"></p>
  <p id="counter_c1"></p>
</body>
</html>
```



and **Preview** . A circle appears in the page with the name "circle one." You can click the square and see the message "You've clicked 1 times on square one," and you can click the circle and see the message "You've clicked 1 times on circle one." Repeated clicks on either object add to the click count in the appropriate displayed sentence.

We added a new **Circle()** constructor that's similar to **Square()**, except that it has a radius, and its **display()** function is different. The **click()** function is exactly the same though.

We also added new code in the **window.onload** function to create a **circle** object, call **setupClickHandler()** to add a click handler to the "c1" <div> object representing the circle in the page, and to call **circle.display()** so we see it in the page.

Each shape gets its own separate click handler. When we pass **square** to **setupClickHandler()**, we set the click handler for the "s1" <div> object to the square's **click()** method, and when we pass **circle** to **setupClickHandler()**, we set the click handler for the "c1" <div> object to the circle's **click()** method.

Because the code for both is exactly the same, we can pull the code out of the two objects and use the same code for both. Modify **this.html** again, as shown.

## CODE TO TYPE:

```
...
function Square(id, name, size) {
    this.id = id;
    this.name = name;
    this.size = size;
    this.numClicks = 0;

    this.display = function() {
        var el = document.getElementById(this.id);
        el.style.width = this.size + "px";
        el.style.height = this.size + "px";
        el.innerHTML = "<p>" + this.name + "</p>";
        console.log(this.name + " has size " + this.size +
            ", and is a " + this.constructor.name);
    };

    this.click = function() {
        this.numClicks++;
        document.getElementById("counter_" + this.id).innerHTML =
            "You've clicked " + this.numClicks + " times on " + this.name;
    };
}

function Circle(id, name, radius) {
    this.id = id;
    this.name = name;
    this.radius = radius;
    this.numClicks = 0;

    this.display = function() {
        var el = document.getElementById(this.id);
        el.style.width = (this.radius * 2) + "px";
        el.style.height = (this.radius * 2) + "px";
        el.style.borderRadius = this.radius + "px";
        el.innerHTML = "<p>" + this.name + "</p>";
        console.log(this.name + " has radius " + this.radius +
            ", and is a " + this.constructor.name);
    };

    this.click = function() {
        this.numClicks++;
        document.getElementById("counter_" + this.id).innerHTML =
            "You've clicked " + this.numClicks + " times on " + this.name;
    };
}

function click() {
    console.log("This in click function: ");
    console.log(this);
    this.numClicks++;
    document.getElementById("counter_" + this.id).innerHTML =
        "You've clicked " + this.numClicks + " times on " + this.name;
}



window.onload = function() {
    var square = new Square("s1", "square one", 100);
    setupClickHandler(square);
    square.display();

    var circle = new Circle("c1", "circle one", 50);
    setupClickHandler(circle);
    circle.display();
}

function setupClickHandler(shape) {
    var elDiv = document.getElementById(shape.id);
    elDiv.onclick = function() {
```

```
        shape.click();
        click.call(shape);
    }
}
...

```

 Save and  Preview. Try clicking on both the square and the circle. You see messages in the page showing how many times you've clicked on the respective shapes. In the console, you see the messages displayed by the new **click()** function we just added, that show which shape you've just clicked on.

First, we removed the **click()** methods from each of the shapes, and put the code into a new **click()** function. The code is exactly the same, except that we added the two console messages at the top. The code in the **click()** method still refers to **this**, and properties like **this.numClicks** and **this.name**, but if you call a global function, **this** is set to the window object. It shouldn't be set to either the **square** or the **circle** though, so what's going on?

This is where **call()** comes in handy. When we set up the click handler for the shapes, we made one small change: we changed the code from **shape.click()** to **click.call(shape)**. So what's the difference? What does **call()** do? Good questions!

**click()** is a function. We can call that function using the **call()** method (I'll explain where that method comes from in a just a moment), and pass in the object that we want to use for **this**, which in this case is the **shape** object, which will be **square** when we've passed **square** to **setupClickHandler**, and **circle** when we've passed **circle** to **setupClickHandler**. Calling the **click()** function using the **call()** method is just like calling the function in the normal way (with **click()**), except that we get to choose how **this** should be defined.

Inside **click()**, **this** is set to either the **square** or the **circle**, depending on which element we click on. If it's set to **square**, we reference the square's **numClicks** property, and the square's **name** property. We do the same for the **circle**.

**call()** is a method of the **click()** function. As we've said previously, a function is an object with properties and methods just like any other object. Whenever you define a function, like **click()**, you're actually creating an instance of the **Function** object using the **Function()** constructor (JavaScript does that for you behind the scenes). Remember that an object can inherit methods and properties from its prototype. In this case, function's prototype includes the method **call()**. You can check for yourself, like this (this session assumes you've loaded the **this.html** file):

```
INTERACTIVE SESSION:

> click
function click() {
  console.log("This in click function: ");
  console.log(this);
  this.numClicks++;
  document.getElementById("counter_" + this.id).innerHTML =
    "You've clicked " + this.numClicks + " times on " + this.name;
}
> click.call
function call() { [native code] }
```

First we ask to display the function **click()**, by typing the name of the function. Then we ask to see the **click()** function's **call** property, which it inherits from its **Function** prototype, and which is implemented natively by the browser (so you can't see the details).

## call() and apply()

The methods **call()** and **apply()** do essentially the same thing, but you use them slightly differently. The first argument of both methods is the object you want to stand in for **this**. If the function you're calling takes arguments, then you also pass these arguments into both **call()** and **apply()**. For **call()**, you pass these arguments as a list of arguments (like you normally do with arguments), and for **apply()**, you pass all the arguments in an array.

So, to use **call()**, you'd write:

OBSERVE:

```
function myFunction(param1, param2, param3) {  
    ...  
}  
var anObject = { x: 1 };  
myFunction.call(anObject, 1, 2, 3);
```

...and to use `apply()`, you'd write:

OBSERVE:

```
function myFunction(param1, param2, param3) {  
    ...  
}  
var anObject = { x: 1 };  
myFunction.apply(anObject, [1, 2, 3]);
```

In both of these examples, the object `anObject` is defined as the value of `this` in the function `myFunction()`, and the arguments `1, 2, 3` are passed to `myFunction()`, and stored in the parameters `param1, param2, and param3`.

As you've seen with our example of using `call()` to call the `click()` function, the arguments are optional—if your function doesn't expect arguments you don't have to supply any.

You can use `call()` and `apply()` on your own functions, as well as JavaScript's built-in functions. For instance, let's say you have an array of numbers and you want to find the maximum number in the array. There is a handy `Math.max()` method available, but it doesn't take an array, it takes a list of numbers:

INTERACTIVE SESSION:

```
> var myArray = [1, 2, 3];  
undefined  
> Math.max(myArray)  
NaN  
> Math.max(1, 2, 3)  
3
```

However, we can use `apply()` to get around this, like this:

INTERACTIVE SESSION:

```
> Math.max.apply(null, myArray);  
3
```

Notice that we pass `null` as the value to be used for `this` (because we don't need any object to stand in for this), and because we're using `apply()`, the values from the array are passed into the `Math.max()` method as a list of arguments.

## Function Arguments

When you invoke a function, you pass arguments to the function's parameters. When the number of arguments is equal to the number of parameters, each parameter gets a corresponding argument from the function call. So, what happens if the number of arguments doesn't match the number of parameters?

Suppose you want to write your own `max()` function (similar to `Math.max()`). How would you do it? Well, you could start simple:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> max </title>
  <meta charset="utf-8">
  <script>
    function max(n1, n2) {
      if (n1 > n2) {
        return n1;
      }
      else {
        return n2;
      }
    }

    console.log(max(99, 101, 103));

  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **max.html**, and **Preview** . In the console, you see 101.

The argument 103 does get passed to the function, but since we don't give it a parameter name, and we don't use it, it's not included in the calculation of the maximum number.

If you pass too *few* arguments, then the parameter that's expecting an argument has the value undefined. Let's see what happens if we pass only one argument to our **max()** function:

**CODE TO TYPE:**

```
function max(n1, n2) {
  console.log("n2 is " + n2);
  if (n1 > n2) {
    return n1;
  }
  else {
    return n2;
  }
}

console.log(max(99, 101, 103));
```



and **Preview** . In the console, you see that "n2 is undefined."


We'd like to write our **max()** function so that it can take any number of arguments, like **Math.max()**, and find the maximum value. To do that, we can use the **arguments** object. This object contains all the arguments passed to a function. Let's rewrite our **max()** function to use it:



#### CODE TO TYPE:

```
function max(n1, n2) {  
  console.log("n2 is " + n2);  
  if (n1 > n2) {  
    return n1;  
  }  
  else {  
    return n2;  
  }  
  var max = Number.NEGATIVE_INFINITY;  
  for (var i = 0; i < arguments.length; i++) {  
    if (arguments[i] > max) {  
      max = arguments[i];  
    }  
  }  
  return max;  
}  
  
console.log(max(99, -55, 101, 103, 22));
```



and . You see the result 103 in the console.

First notice that our **max()** function no longer specifies any parameters. That's because we're going to access all the arguments with the **arguments** object.

The **arguments** object is an array-like object; you can iterate over it, and it has a length property, so we can use it like an array to iterate over all of the arguments and find the maximum value. To do that we initialize the variable **max** to negative infinity, and then look at each argument to see if it's greater than **max**. Each time an argument is greater than **max**, we update the value of **max**, so the final result is the maximum value of all the arguments we passed into the object.

## The Four Ways to Invoke a Function

You've seen examples of each of the four ways you can invoke a function:

- as a function
- as a method
- as a constructor
- with `apply()` or `call()`

You'll likely use the first three most often, but there are times when the fourth (using **apply()** or **call()**) can come in handy too.

In general, the **this** keyword is bound to the object that contains the function. For global functions, that's the global window object; for constructors, that's the object being constructed; and for methods, that's the object that contains the method you're calling.

Three big exceptions to this rule are: when you are in an event handler function on an event like "click"; when you've explicitly changed the value defined for **this** using **call()** or **apply()**; and when you are in a nested function. Memorize the way **this** behaves in these three cases so you don't get tripped up (not to mention that having a grasp on this is a great way to ace those JavaScript interview questions!)

In this lesson, you learned about the four ways we can invoke functions in JavaScript, and what happens to **this** in each case. Take some time to practice invoking some functions before you move on to the next lesson, where we'll look at invocation patterns: code designs that use function calls in some interesting ways.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Invocation Patterns

---

## Lesson Objectives

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

- call a function recursively.
  - create an object so its methods can be chained.
  - call functions by chaining them together.
  - create a static method.
  - distinguish between static and instance methods.
- 

## Invocation Patterns

In this lesson, we take a look at some "invocation patterns": that is, ways to structure your function calls. These aren't new ways to invoke functions, but rather code designs involving function calls that may be useful as you continue in your JavaScript programming.

### Recursion

Recursion is when you call a function from within that same function. It's a powerful programming tool, so it's an important concept to master, but it can be tricky.

A recursive function can always be converted to an iteration, so we'll start by looking at an example of iteration and then rewrite the function using recursion instead.

We'll use code from a previous example using the **Square()** constructor to create squares in the page. Create a new HTML file and copy in this code:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Recursion </title>
  <meta charset="utf-8">
  <style>
    .square {
      background-color: lightblue;
      cursor: pointer;
    }
    .square p {
      padding-top: 35%;
      text-align: center;
      -webkit-user-select: none;
      -moz-user-select: none;
      -ms-user-select: none;
      user-select: none;
    }
  </style>
  <script>
    function Square(id, size) {
      this.id = id;
      this.size = size;

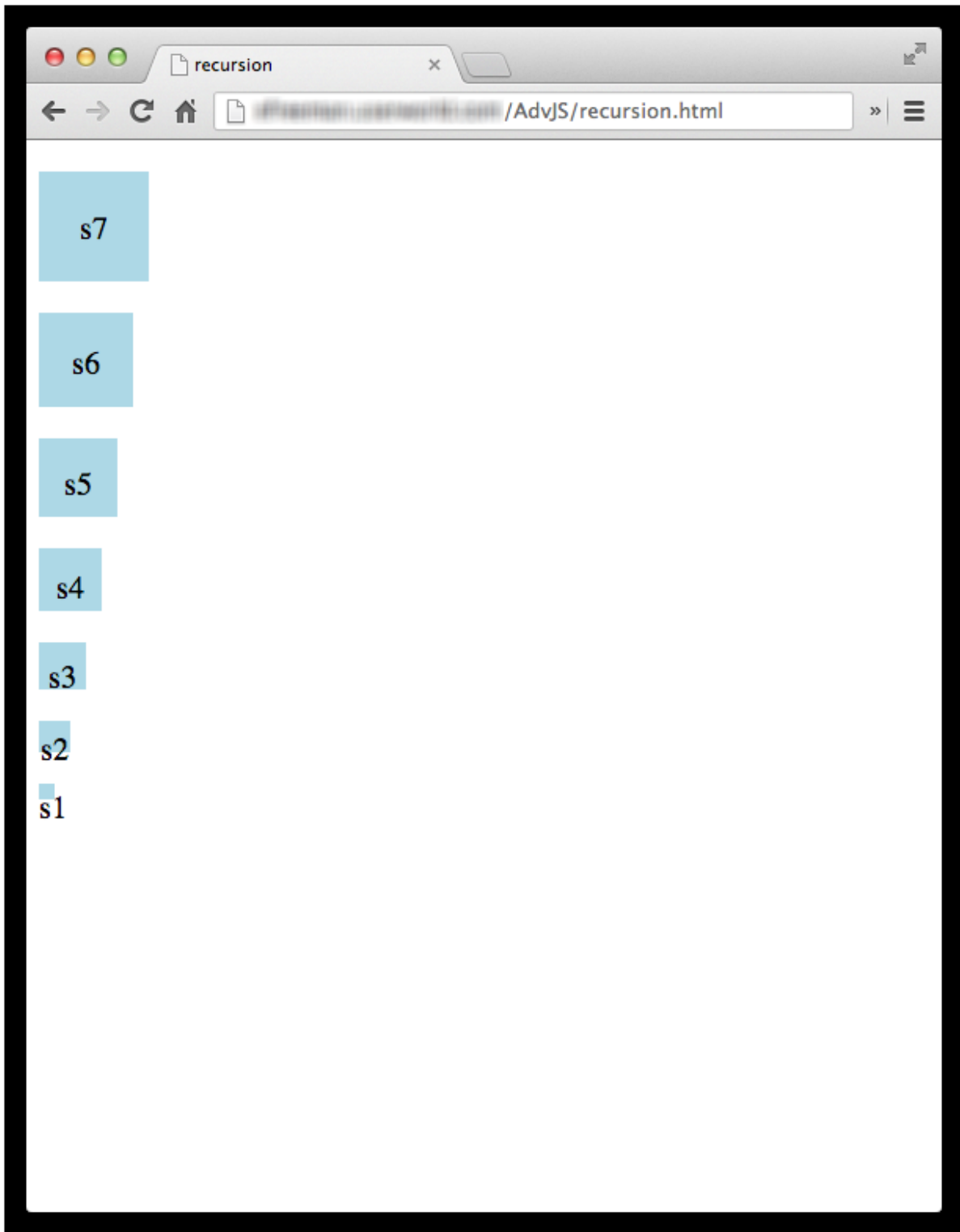
      this.display = function() {
        var el = document.createElement("div");
        el.setAttribute("id", this.id);
        el.setAttribute("class", "square");
        el.style.width = this.size + "px";
        el.style.height = this.size + "px";
        el.innerHTML = "<p>" + this.id + "</p>";
        console.log(this.id + " has size " + this.size +
          ", and is a " + this.constructor.name);
        document.getElementById("squares").appendChild(el);
      };
    }

    function createSquares(n) {
      var size = 10;
      if (n == 0) {
        return;
      }
      while (n >= 1) {
        var s = new Square(("s" + n), n * size);
        s.display();
        n--;
      }
    }

    window.onload = function() {
      createSquares(7);
    };
  </script>
</head>
<body>
<div id="squares"></div>
</body>
</html>
```



Save this in your **/AdvJS** folder as **recursion.html**, and  **Preview**. You see seven squares, decreasing in size:



(The names of the smallest squares don't fit into the `<div>s` properly; don't worry about that).

Take a look at the code:

**OBSERVE:**

```
function createSquares(n) {
  var size = 10;
  if (n == 0) {
    return;
  }
  while (n >= 1) {
    var s = new Square(("s" + n), n * size);
    s.display();
    n--;
  }
}
```

We use a function, **createSquares()**, to create a given number of squares. In a **loop**, we invoke the **Square()** constructor, and then display the resulting **square** object by calling its **display()** method.

The **display()** method creates a new <div> object representing the square, and appends it to the "squares" <div> in the page.

To create the correct number of squares, we use a **while loop** to iterate the number, **n**, passed into **createSquares()**. We also use **n** to compute the size of the square to add to the page (using **n \* size**, where size is 10), and include it as part of the name of each square (for example, s3).

We can rewrite this function using *recursion* by replacing the while loop with a call to the function **createSquares()**:

**CODE TO TYPE:**

```
...
function createSquares(n) {
  var size = 10;
  if (n == 0) {
    return;
  }
  while (n >= 1) {
    var s = new Square(("s" + n), n * size);
    s.display();
    n--;
  }
  var s = new Square(("s" + n), n * size);
  s.display();
  createSquares(n-1);
}
...
```



and **Preview** . You see the same seven squares in decreasing sizes.

Let's compare the recursive version to the version with the iteration to see how that works:

**OBSERVE:**

```
while (n >= 1) {
  var s = new Square(("s" + n), n * size);
  s.display();
  n--;
}
```

Here we **iterate** through all of the values of **n** until **n** is equal to 1. When we pass 7 to the function **createSquares()**, the first time through the iteration, **n** is 7, we get a square of size 70 (using **n \* size**, which is 7 \* 10), then reduce **n** by one, and keep looping until **n** is 1.

**OBSERVE:**

```
var s = new Square(("s" + n), n * size);
s.display();
createSquares(n-1);
```

The recursive version does essentially the same thing, except that we call **createSquares()** each time we want a square of a smaller size.


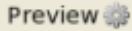
The first time we call **createSquares()**, **n** is 7, so we create and display a square of size 70. Then we call **createSquares()** again, only we pass **n - 1** as the argument to the function. So in this call to **createSquares()**, **n** is 6. We create and display a square of size 60, and call **createSquares()** again, with the argument 5, and so on.

When **n** is 1, we execute the code that creates and displays a square of size 10. Then we call **createSquares()** and pass the value 0 for **n**. Now, instead of executing the code to create a square of size 0, we check and see that **n** is 0 and return:

**OBSERVE:**

```
if (n == 0) {
    return;
}
```

This if statement is known as the *base case* and it's vitally important because without it, the recursion will

continue forever—try it. Comment out the **if** block, , and . It's just like with iteration: you must supply a conditional test to tell the iteration when to stop. In recursion, the base case tells the recursion when to stop. We want the recursion to stop when **n** is 0. So in this case, we do *not* call **createSquares()** again, which causes the recursion to stop.

The results of both of these versions of **createSquares()** are exactly the same, but the first uses iteration and the second uses recursion. Again, recursion can always be replaced with iteration.

## Why Use Recursion?

Some algorithms are naturally recursive. For instance, the algorithm to compute the factorial of a number **n** is recursive. To compute the factorial of a number, say 5, we multiply 5 times the factorial of that number minus 1:

**OBSERVE:**

```
The factorial of 5 is 5 * the factorial of 4.
The factorial of 4 is 4 * the factorial of 3.
The factorial of 3 is 3 * the factorial of 2.
The factorial of 2 is 2 * the factorial of 1.
The factorial of 1 is 1.
So the factorial of 5 is 5 * 4 * 3 * 2 * 1, which is 120.
```

Writing this in pseudocode, you can think of this as recursively designed code:

**OBSERVE:**

```
factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
```

We are using the factorial function in the definition of the factorial function. That's the very definition of "recursive." It's like we are using the question in the answer to the question: what is factorial of 5? It's 5 times the factorial of 4!

Of course, that can be frustrating unless you have an answer to the question that doesn't involve the question itself. That's the reason for the base case. The base case for the factorial algorithm is 1; when **n** is 1, we don't call factorial again. That means you can finally stop asking the question and start getting answers. You can

plug in the value 1 for the answer to "What is the factorial of 1?" then you can then answer the question, "What is the factorial of 2?" and so on until you get to the answer for your original question, "What is the factorial of 5?"

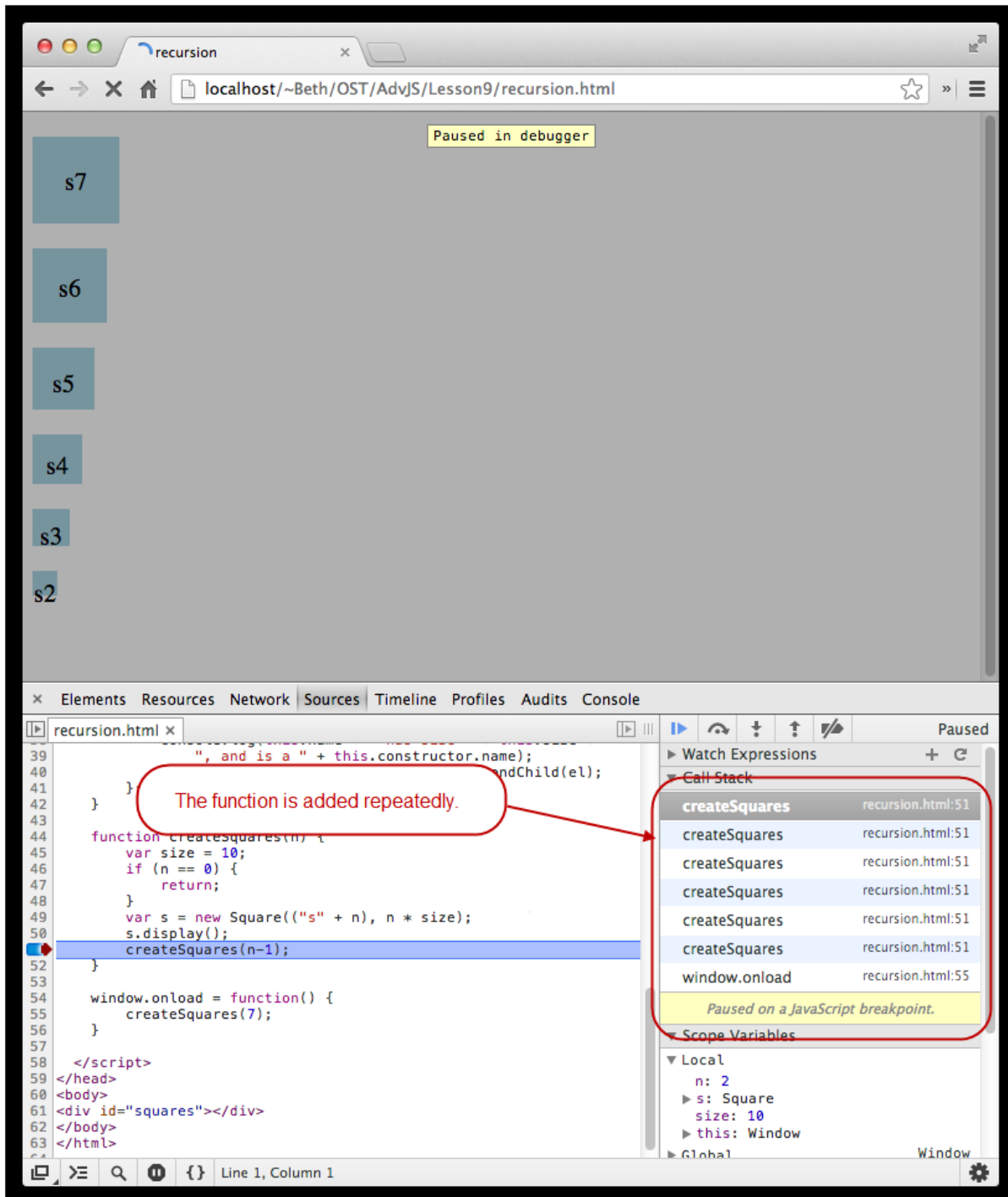
Every recursive algorithm works this way: you create a pile of function calls, each with solutions that involve calling that function again, until you get to the base case. Then you can start unravelling the pile until you get back to your original function call. (See if you can implement **factorial()** in JavaScript—it'll be one of the projects!)

Many algorithms for which you may want to create functions are naturally recursive. These naturally recursive functions tend to be easier to read when they are expressed recursively, rather than when they are expressed iteratively (using loops).

However, there's a downside to recursion. Each time you call a function, you add that function to the call stack; this takes up memory. Iteration takes up memory too, but usually not as much as a pile of functions on a call stack. To see the call stack created by recursive calls, we can use the Chrome browser tools and add a breakpoint to the code on the line in **createSquares()** where we call **createSquares()** recursively:

The screenshot shows a web browser window with a page titled "recursion" at the URL "http://localhost:3000/AdvJS/recursion.html". The page content consists of seven blue square elements stacked vertically, labeled "s1" through "s7" from bottom to top. Below the browser window, the Chrome DevTools interface is open to the "Sources" tab, showing the source code for "recursion.html". The code defines a `createSquares(n)` function that recursively creates squares. A red circle highlights line 51, which is `createSquares(n-1);`, and a red arrow points to it with the text "Set a breakpoint here." in a red oval. The right-hand sidebar of DevTools shows the "Breakpoints" panel with a checked breakpoint at "recursion.html:51 createSquares(n-1);". The "Call Stack" and "Scope Variables" panels are also visible, both showing "Not Paused".

Then we reload the page, and execution stops each time we call `createSquares()` recursively. Execute the breakpoint a few times by clicking the **Resume script execution** button, and you can see the function being added to the call stack each time we call it:



Take a look at the scope variables each time you click the **Resume script execution** button; the value of `n` decreases by one each time. Eventually, `n` gets to 0, the recursion stops, and the code completes.

This happens because we're calling `createSquares()` from *inside* `createSquares()` *before* the previous invocation of `createSquares()` is complete. To compare, let's say you have a function `add()` (that's not recursive), and you call that function three times:



**OBSERVE:**

```
function add(num1, num2) {  
    return num1 + num2;  
}  
add(1, 2);  
add(2, 3);  
add(3, 4);
```

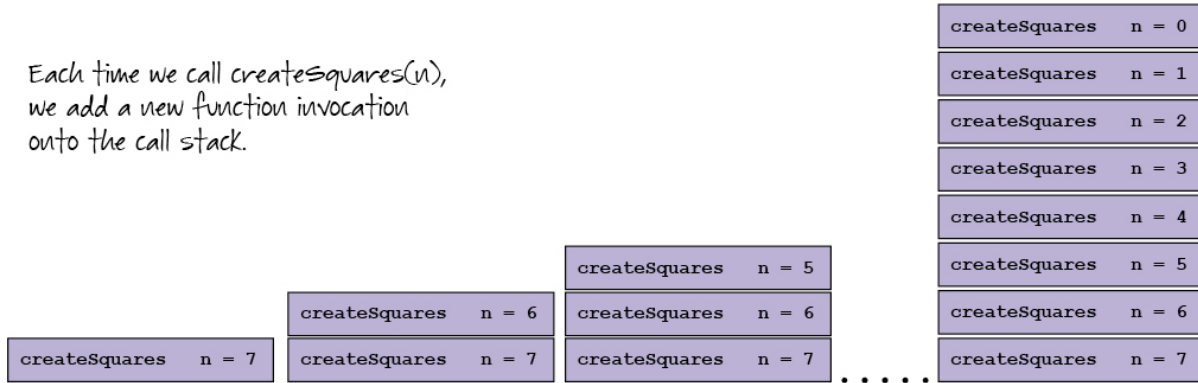
The **add()** function goes on to the call stack three times. However, you only have *one* invocation of **add()** on the call stack at a time, because each invocation of **add()** ends before the next one begins, so the call stack never gets bigger than one function.

Now think about **createSquares()** again. We call **createSquares()** *before* the previous call to **createSquares()** has completed. So we call **createSquares(7)**, and while that function invocation is still on the stack, we call **createSquares(6)**. This function invocation goes on top of the invocation to **createSquares(7)**, so we have *two* function invocations on the stack. Then we do it again with **createSquares(5)**, and so on until we call **createSquares(0)**.

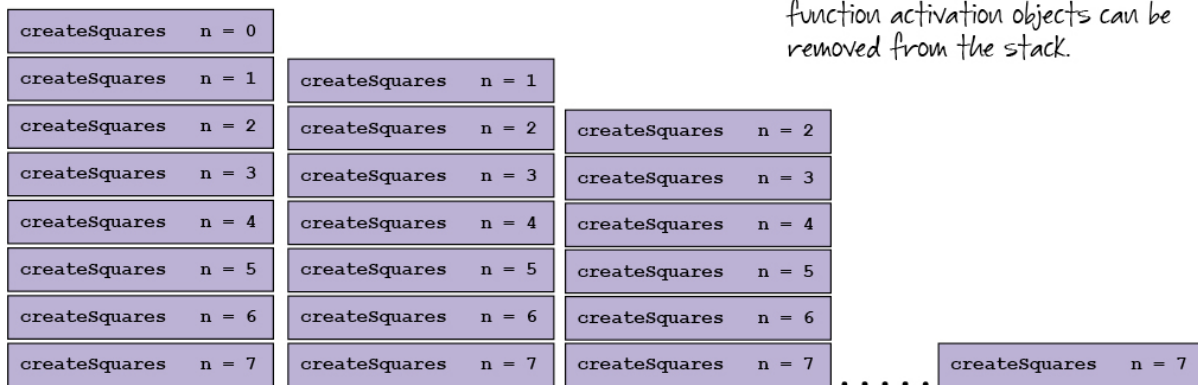
When we call **createSquares(0)**, **createSquares(0)** gets added to the top of the stack, but **createSquares(0)** just returns, so it gets popped off the stack right away. When **createSquares(0)** returns, then **createSquares(1)** can finish executing and then get popped off the stack. Once that's done, **createSquares(2)** can finish executing and get popped off the stack, and so on, until finally, **createSquares(7)** finishes and gets popped off the stack and you're done.

Here's how the call stack gets built up as each recursive call to **createSquares()** takes place. Then the function invocations are removed from the stack once we reach the base case and each function call can finish:

Each time we call `createSquares(n)`, we add a new function invocation onto the call stack.



After the "base case" when `createSquares(0)` returns 0, each invocation can finish, so the function activation objects can be removed from the stack.



Once the last function invocation has completed, we can see all our squares in the page.

This pile of function invocations that's created on the call stack when you call a function recursively isn't a big deal if your functions are small (and don't have too many variables in each activation object), and if the number of times the function is called recursively (so the number of invocations that goes on the stack) is small. If your functions are large though and have lots of variables and/or the function is called recursively many times, then your code will take up a lot of memory. In addition, your browser will limit the number of functions it allows on the call stack at once.

Most of the time, you'll have functions with few local variables, so you won't call your recursive functions so often that it becomes problematic. So now you know what to look out for if you run into memory issues while using a recursive design.

## Chaining (a la jQuery)

Another invocation pattern you'll see in JavaScript is method *chaining*. Chaining is common in some JavaScript libraries, like jQuery. The technique allows you to write multiple method calls on the same object in a chain. For instance, instead of writing:

```
OBSERVE:
obj.method1 ();
obj.method2 ();
obj.method3 ();
```

...you can write:

OBSERVE:

```
obj.method1().method2().method3();
```

For method chaining to work, each method must return an object so that the next method can be called on that object. In the example above, if **obj.method1()** returns **obj**, that object is used to call **method2()** immediately.

Let's take a look at a concrete example. We'll use the same Squares example from earlier, and modify it a bit. Save **recursion.html** to a new file in your **/AdvJS** folder named **chaining.html**, and make these changes:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Chaining </title>
  <meta charset="utf-8">
  <style>
    .square {
      background-color: lightblue;
      cursor: pointer;
    }
    .square p {
      padding-top: 35%;
      text-align: center;
      -webkit-user-select: none;
      -moz-user-select: none;
      -ms-user-select: none;
      user-select: none;
    }
  </style>
  <script>
    function Square(id, size) {
      this.id = id;
      this.size = size;
      this.el = null;

      this.bigger = function(size) {
        if (this.el) {
          this.size += size;
          this.el.style.width = this.size + "px";
          this.el.style.height = this.size + "px";
          return this;
        }
      };

      this.color = function(color) {
        if (this.el) {
          this.el.style.backgroundColor = color;
          return this;
        }
      };

      this.display = function() {
        var this.el = document.createElement("div");
        this.el.setAttribute("id", this.id);
        this.el.setAttribute("class", "square");
        this.el.style.width = this.size + "px";
        this.el.style.height = this.size + "px";
        this.el.innerHTML = "<p>" + this.id + "</p>";
        console.log(this.id + " has size " + this.size +
          ", and is a " + this.constructor.name);
        document.getElementById("squares").appendChild(this.el);
        return this;
      };
    }

function createSquares(n) {
  var size = 10;
  if (n == 0) {
    return;
  }
  var s = new Square(("s" + n), n * size);
  s.display();
  createSquares(n - 1);
}

    window.onload = function() {
```

```

        createSquares(7);

        var mySquare = new Square("mySquare", 100);
        mySquare.display().color("green");
        mySquare.el.onclick = function() {
            mySquare.bigger(50).color("red");
        };
    };
</script>
</head>
<body>
<div id="squares"></div>
</body>
</html>

```



Save and **Preview**. You see a green square with the name "mySquare." Click on the square. The square gets bigger and turns red. Each time you click on the square it gets bigger.

In our code, we made a modification to the **display()** method: now that method stores the <div> element we're creating for the square in the **Square** object, in the property **el**. Now we can use that <div> element in the two new methods we've added: **bigger()** and **color()**. **bigger()** takes a value and adds it to the size of the square, then modifies the style of the <div> representing the square to change the size of the <div> (which makes the square appear bigger). Similarly, **color()** takes a color string (like "blue," "green," or "red"), and modifies the style of the <div> to change the background to that color.

We've also added a line to each method to return **this**. For example:

OBSERVE:

```

this.color = function(color) {
    if (this.el) {
        this.el.style.backgroundColor = color;
        return this;
    }
};

```

Each of the **Square**'s three methods now returns **this**, so when we call **mySquare.display()**, we get **mySquare** back as a result. That means we can call one of the other methods on the resulting object immediately, like **color()** or **bigger()**. We can *chain* these methods together!

In fact, that's exactly what we do in the code after we create **mySquare**:

OBSERVE:

```

mySquare.display().color("green");
mySquare.el.onclick = function() {
    mySquare.bigger(50).color("red");
};

```

After creating the **mySquare** object, **we call the display() method** to create and display the <div> object in the page, and then **chain a call to the color() method** to turn the square green. Also, we set up a click handler on the <div> so that when you click on the square, we call two methods on the square, **bigger() and color(), again chained**, so that **color()** is called on the object that is returned by **bigger()**.

Usually when you chain methods, you call methods on the same object for each part of the chain. It's certainly possible to have one of the methods return a different object for the next method to be called on, but code written that way would be much more difficult to understand, so in general, it's not recommended. When you create method chains, you generally want to make sure you act on the *same* object in each part of the chain.

Potential benefits of chaining are that it can make code easier to read, and reduce the number of lines of code. However, chains that are too long can be difficult to understand as well, so use chaining judiciously. (Chains that are too long are often called *train wrecks!*)

## Static vs. Instance Methods

One last invocation pattern we'll look at in this lesson is how to call static methods, and the differences

between static and instance methods.

Let's begin by exploring the **Date()** constructor. You can use **Date()** to create new date objects, like this:

#### INTERACTIVE SESSION:

```
> var nowDate = new Date()
undefined
> nowDate.toString()
"Thu Sep 05 2013 10:17:39 GMT-0700 (PDT)"
> nowDate.getMonth()
8
> nowDate.getTime()
1378401459465
```

Here, we created a new date object, **nowDate**, by calling the **Date()** constructor function. If you pass no arguments to the constructor, this function creates an object for the current date and time, so the result is a date object that represents "right now," which (as of the writing this lesson) is Thursday, September 5, 2013 at 10:17am.

The date object **nowDate** has various methods and properties you can use just like any other object. For instance, you can use the method **nowDate.toString()** to get a string representing the current date and time, and get the month with **nowDate.getMonth()** (note that the returned numeric value is from an array whose indices start at zero, so September is represented by 8). You can get the number representing the date and time using the **getTime()** method.

Now try this:

#### INTERACTIVE SESSION:

```
> var time = Date.now()
undefined
> time
1378401746077
```

We called a method **now()** on the **Date** object. **Date** and **Date()** are the same object: they are both the **Date()** constructor function. Remember that functions are objects—and objects can have properties though. The method **now()** is a method of the **Date()** function object.

Notice that the result, **time**, is a variable that contains a number representing the current time. It's different from the variable **nowDate** though. **nowDate** is a **Date** object, whereas **time** is simply a number:

#### INTERACTIVE SESSION:

```
> nowDate instanceof Date
true
> time instanceof Date
false
```

You can use **time** to create a **Date** object, like this:

#### INTERACTIVE SESSION:

```
> var anotherDate = new Date(time);
undefined
> anotherDate
Thu Sep 05 2013 10:22:26 GMT-0700 (PDT)
> anotherDate.getTime()
1378401746077
```

Once you have the **anotherDate** object, you can use the method **getTime()** to get the number representing the date and time back (it's the same number as in the variable **time** that we used to create the object in the first place).

So, what's the difference between creating a date object using a constructor and then calling methods on that date object, and calling a method directly on the function object itself? And how do you add methods directly to a function anyway?

We'll answer those questions by adding a method to the **Square()** constructor we've been working with in this lesson. Modify your JavaScript code in **chaining.html** as shown:

#### CODE TO TYPE:

```
function Square(id, size) {
  this.id = id;
  this.size = size;
  this.el = null;

  this.bigger = function(size) {
    if (this.el) {
      this.size += size;
      this.el.style.width = this.size + "px";
      this.el.style.height = this.size + "px";
      return this;
    }
  };

  this.color = function(color) {
    if (this.el) {
      this.el.style.backgroundColor = color;
      return this;
    }
  };


  this.display = function() {
    this.el = document.createElement("div");
    this.el.setAttribute("id", this.id);
    this.el.setAttribute("class", "square");
    this.el.style.width = this.size + "px";
    this.el.style.height = this.size + "px";
    this.el.innerHTML = "<p>" + this.id + "</p>";
    console.log(this.id + " has size " + this.size +
      ", and is a " + this.constructor.name);
    document.getElementById("squares").appendChild(this.el);
    return this;
  };
}

Square.info = function() {
  return "Square is a constructor for making square objects with an id and
size.";
};

window.onload = function() {
  var mySquare = new Square("mySquare", 100);
  mySquare.display().color("green");
  mySquare.el.onclick = function() {
    mySquare.bigger(50).color("red");
  };

  var info = Square.info();
  console.log(info);
};
```



and . In the console, the message, "Square is a constructor for making square objects with an id and size." is displayed.

We're working with two different kinds of objects here. First, we use the **Square()** constructor to create new objects by calling **Square()** with **new**:

OBSERVE:

```
var mySquare = new Square("mySquare", 100);
```

In this case, the object **mySquare** is called an *instance* of **Square**: it's an object created by the constructor, an object that has the properties and methods we specify in the constructor by saying **this.PROPERTYNAME = SOME VALUE**. Methods like **bigger()**, **color()**, and **display()** are called *instance methods*, because they are methods of the object instances created by calling **Square()** with **new**.

The other object we're working with is the **Square** object. This object happens to be a function, but it's like other objects in that it has methods and properties. Also, just like any other object, we can add a new property or method to it:

OBSERVE:

```
Square.info = function() {  
    return "Square is a constructor for making square objects with an id and size."  
};
```

In this case, we're adding a method of the **Square** object, *not* a method of the **mySquare** instance object we created using **Square()** as a constructor. We call methods like this *static methods*: they are methods of the constructor function object. We say they are "static" because, unlike instance methods that may return different values depending on the properties of the specific instance you're working with (for example, one square might have size 10 and color red, while another might have size 200 and the color green), static methods don't vary based on those differences. Static methods are methods of the constructor, *not* methods of the instances.

We can't access static methods from object instances, just like we can't access instance methods from the constructor object. Let's test this:

CODE TO TYPE:

```
window.onload = function() {  
    var mySquare = new Square("mySquare", 100);  
    mySquare.display().color("green");  
    mySquare.el.onclick = function() {  
        mySquare.bigger(50).color("red");  
    };  
  
    var info = Square.info();  
    console.log(info);  
  
    mySquare.info();  
}
```



and **Preview**. In the console, you see the error:

OBSERVE:

```
Uncaught TypeError: Object #<Square> has no method 'info'
```

Here, we try to call a static method, **info()**, from an object instance. It doesn't work.

Try this:



**CODE TO TYPE:**

```

window.onload = function() {
  var mySquare = new Square("mySquare", 100);
  mySquare.display().color("green");
  mySquare.el.onclick = function() {
    mySquare.bigger(50).color("red");
  };

  var info = Square.info();
  console.log(info);

  mySquare.info();
  Square.color("red");
};

```



and **Preview**. In the console, you see another error message:

**OBSERVE:**

```

Uncaught TypeError: Object function Square(id, size) {
  ...
} has no method 'color'

```

Here, we try to access an instance method from the **Square** object, and again, it won't work.

Let's go back to the **Date** object. When we created a new date object, **nowDate**, then called methods on that object, like **getMonth()** and **getTime()**, we used *instance methods*: methods defined in the instance objects we create by calling the **Date()** constructor function with **new**. When we called **Date.now()**, we used a *static method*: a method defined in the **Date** function object itself.

So, when should a method be an instance method, and when should a method be a static method?

Well, object instances are used to represent specific items, like a square object with a specific size, or a date object with a specific date and time. So it makes sense to have methods like **getMonth()** and **getFullYear()** for a date object *instance*, because that object has specific values for the month and the year, the values you gave it when you created the object using the constructor with **new**.

The constructor function, **Date()** doesn't represent a specific date or time. So asking the **Date** object for a month, for instance:

**OBSERVE:**

```
Date.getMonth()
```

...makes no sense. **Date** doesn't represent any particular date until you create a specific instance. However, the **Date** object *can* have useful properties and methods that aren't related to a specific date you're creating. The **now()** method is one of those useful methods: it generates a number that represents the date and time of "right now." There is no need to create a new date object instance to get that number.

Similarly, the **Square.info()** method provides information about the **Square** constructor function that is not related to any specific instance of a square. Meanwhile, methods like **color()** and **display()** are only relevant for a specific instance of a square, one that has a size and an element associated with it that can be assigned a color.

So, in a sense, invoking static methods and instance methods are the same. A static method is invoked on the object in which it is defined (the constructor function) and an instance method is invoked on the object in which it's defined as well (the instance object created by calling the constructor function). The trick is to keep track of which object is which.

Further, we don't just use *static* and *instance* to describe methods; we can also use these terms to describe other properties of objects. So, in our Squares example, we say the property **size** is an *instance variable* of a square object. You could add a *static variable*, say, **recommendedSize**, to the **Square()** constructor, like this:

OBSERVE:

```
Square.recommendedSize = 100;
```

To access the variable from the **info()** method, you'd write:

OBSERVE:

```
Square.info = function() {  
  return "Square is a constructor for making square objects with an id and siz  
e. " +  
      "The recommended size for a square is " + Square.recommendedSize + "  
";  
};
```

In this lesson, we talked about three invocation patterns that you'll see used frequently in JavaScript: recursion, chaining, and static methods. There are other invocation patterns, of course, which you may encounter as you continue your JavaScript studies and get more programming experience. In upcoming lessons we'll explore the Module Pattern, but for now, do the quiz and the project, then take a break. When you're rested, continue on!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Encapsulation and APIs

---

## Lesson Objectives

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

- create private properties and methods in an object.
  - create a public interface to use the object.
- 

## Encapsulation and APIs

One of the benefits of objects in an object-oriented language is encapsulation: the ability to compartmentalize properties and behaviors in an object, and provide an interface so the rest of your program doesn't have to worry about how certain behaviors are implemented. The object just works. Think of other objects you use in JavaScript, like JSON or Date or Math. These objects all have properties and methods that work; you don't need to worry about the inner workings. Sometimes you might want to hide the internal operation of your properties and methods, so programs don't depend on any particular implementation. In this lesson, we'll learn how to use objects and functions to encapsulate structure and behavior, and how to use information hiding techniques to protect implementation details.

### Privacy, Please

In the previous lesson, we used a **Square()** constructor to create square objects and display them in a web page. In that example, we created some properties and methods to give a new square object an id, a size, and a <div> element to represent the square object in the web page:

**OBSERVE:**

```
function Square(id, size) {
  this.id = id;
  this.size = size;
  this.el = null;

  this.bigger = function(size) {
    if (this.el) {
      this.size += size;
      this.el.style.width = this.size + "px";
      this.el.style.height = this.size + "px";
      return this;
    }
  };

  this.color = function(color) {
    if (this.el) {
      this.el.style.backgroundColor = color;
      return this;
    }
  };

  this.display = function() {
    this.el = document.createElement("div");
    this.el.setAttribute("id", this.id);
    this.el.setAttribute("class", "square");
    this.el.style.width = this.size + "px";
    this.el.style.height = this.size + "px";
    this.el.innerHTML = "<p>" + this.id + "</p>";
    console.log(this.id + " has size " + this.size +
      ", and is a " + this.constructor.name);
    document.getElementById("squares").appendChild(this.el);
    return this;
  };
}

window.onload = function() {
  var mySquare = new Square("mySquare", 100);
  mySquare.display().color("green");
  mySquare.el.onclick = function() {
    mySquare.bigger(50).color("red");
  };
};
```

All of the values in a square object are either properties or methods, which means we could change them at any time. For instance, you could write:

**OBSERVE:**

```
mySquare.size = 200;
```

...and that would change the **size** property of the **mySquare** object, but the square in the web page wouldn't get any bigger.

Just changing the **size** property of a square doesn't affect the <div> element (stored in the **el** property) at all. Unless you call the **bigger()** method of a square, no changes will be made to the size of the <div> in the page.

You could also do this:

**OBSERVE:**

```
mySquare.el = document.createElement("p");
```

...because if the element that represents the square isn't set up correctly (by calling the **display()** method), the

square won't appear in the page.

Now, *you* probably wouldn't make these mistakes, but if you give your **Square** code to a friend, and your friend doesn't quite understand how to use it correctly, all kinds of things could go wrong.

So developers like to create objects that *protect* an object's inner workings, and provide a simple *interface* for other developers to use to manipulate the object. An interface is a *public* view of an object that hides its inner workings, but lets the person using the object work with it.

For instance, we might want to allow someone to create a square, but rather than controlling the display of the square, have the square handle that privately so that a square is only displayed once. (Right now, you could call **display()** twice.) We might want to protect the **e1** property so that no one can change it. We might want to control the color and the amount by which a square grows each time internally within the square object, and only allow the user of a square object to "grow" the square.

## An Example

That's pretty abstract, so let's take a look at an example. We'll modify our **Squares** example, but start from scratch with a new file. Go ahead and create a new file and add this code:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Squares with API </title>
  <meta charset="utf-8">
  <style>
    .square {
      background-color: lightblue;
      cursor: pointer;
    }
    .square p {
      padding-top: 35%;
      text-align: center;
      -webkit-user-select: none;
      -moz-user-select: none;
      -ms-user-select: none;
      user-select: none;
    }
  </style>
  <script>
    function Square(size) {
      var initialSize = size;
      var el = null;
      var id = getNextId();

      this.grow = function() {
        setBigger(10);
        setColor("red");
      };

      var self = this;
      display();

      function setBigger(growBy) {
        if (el) {
          size += growBy;
          el.style.width = size + "px";
          el.style.height = size + "px";
        }
      }

      function setColor(color) {
        if (el) {
          el.style.backgroundColor = color;
        }
      }

      function display() {
        el = document.createElement("div");
        el.setAttribute("id", id);
        el.setAttribute("class", "square");
        el.style.width = size + "px";
        el.style.height = size + "px";
        el.innerHTML = "<p>" + id + "</p>";
        el.onclick = self.grow;
        document.getElementById("squares").appendChild(el);
      }

      function getNextId() {
        var squares = document.querySelectorAll(".square");
        if (squares) {
          return squares.length;
        }
        return 0;
      }
    }
  </script>
</head>
<body>
  <div id="squares">
    <div class="square">
      <p>0</p>
    </div>
  </div>
</body>
</html>
```

```


    }

    window.onload = function() {
        var square1 = new Square(100);
        var square2 = new Square(200);

        var growButton = document.getElementById("growButton");
        growButton.onclick = function() {
            square1.grow();
            square2.grow();
        };
    };
</script>
</head>
<body>
<form>
    <input type="button" id="growButton" value="Grow">
</form>
<div id="squares"></div>
</body>
</html>

```



Save this in your **/AdvJS** folder as **squaresAPI.html**, and **Preview** . When you click on a square, that square grows; when you click on the **Grow** button, both squares grow.

Let's take a closer look at the code.

#### OBSERVE:

```

function Square(size) {
    var initialSize = size;
    var el = null;
    var id = getNextId();

    this.grow = function() {
        setBigger(10);
        setColor("red");
    };

    ...

window.onload = function() {
    var square1 = new Square(100);
    var square2 = new Square(200);

    var growButton = document.getElementById("growButton");
    growButton.onclick = function() {
        square1.grow();
        square2.grow();
    };
};

```

First, notice that we now have only one *public* property in the **Square()** constructor: a method **grow()**. What do we mean by "public" here? Well, think back to how constructors work. When you call a function like **Square()** with **new**, the constructor creates a new object instance with any properties and methods you add to it using **this** in the constructor. In this example, we add only *one* property to the object being created by the **Square()** constructor: the **grow()** method. So, that's the only property the resulting object will contain.

So what happens to all that other stuff after the object's created? How can we use the variables and the functions in the square object if they go away after the **Square()** constructor has finished executing?

Those are really great question. We'll answer them in detail in the next lesson. The short answer is: a closure. Some of the other "stuff" in the constructor is accessible after **Square()** is complete and has returned a new square object because it's saved in a closure. Don't worry about that now; just keep it in the back of your

mind, and know that the variables and functions encapsulated with the object are available even after the constructor has completed.

## Private Variables

All the values that we created as properties before are now variables:

```
OBSERVE:
function Square(size) {
  var initialSize = size;
  var el = null;
  var id = getNextId();
  ...
}
```

These variables are *private*. You can't access them by writing something like:



```
OBSERVE:
square1.id
```

...in your code that creates the square objects. Try it. See what happens if you try to access one of these variables.

```
CODE TO TYPE: Update your code in squaresAPI.html to add the following JavaScript code
...
window.onload = function() {
  var square1 = new Square(100);
  var square2 = new Square(200);

  console.log(square1.id);

  var growButton = document.getElementById("growButton");
  growButton.onclick = function() {
    square1.grow();
    square2.grow();
  };
};
```

 and  and check the console: "undefined." **square1** doesn't have an **id** property, so the value of **square1.id** is undefined.

Once you've tested this code, go ahead and remove the line you added.

These variables are now accessible only *inside* the constructor function. They are not accessible by users of the square objects, so we say they are *private*. We use the **initialSize** variable to keep track of the initial size of the square (we can change the size of the square using **this.grow()**, so we might want to know what the original size was in case we need it later). We'll use the **el** variable to hold the `<div>` element (once we create it in the **display()** function) and **id** to hold a unique id for the square, which we'll generate using **getNextId()**.

## Private Functions

The **Square()** constructor doesn't take an id (unlike the version in the previous lesson); instead, we generate an id in the constructor based on how many squares are in the page already. We use a nested function, **getNextId()**, to generate the next id. Let's examine this function more closely:



#### OBSERVE:

```
function Square(size) {
  var initialSize = size;
  var el = null;
  var id = getNextId();

  ...

  function getNextId() {
    var squares = document.querySelectorAll(".square");
    if (squares) {
      return squares.length;
    }
    return 0;
  }
};
```

To set the value of the `id` variable to the next id (that is, a number that is no longer being used as an id by any of the existing squares), we call `getNextId()`. This function first **gets all the existing elements with the class "square"** from the HTML page using `document.querySelectorAll()`, which then returns an array of elements (if any exist). If we have no squares yet, the array will be empty, and so the length is 0, and the next id should be 0. Similarly, if we have one square in the page, then the length of the array is 1, and the next id should be 1, and so on.

Again, note that the function `getNextId()` is not accessible to code that creates and uses square objects. This nested function is *private* and can be accessed only within the constructor function.

We set the id of the square as it's being created when the constructor function runs, and we don't need `getNextId()` at all after the object has been created.

## A Public Method

So far, the variables (`initialSize`, `el`, and `id`), and the function (`getNextId()`) that we've looked at have all been *private*. The next function, `grow()`, is a public method of the object. That means that the method is added to the object created by the constructor, and that the code that creates and uses the square object can call the method.

#### OBSERVE:

```
function Square(size) {
  var initialSize = size;
  var el = null;
  var id = getNextId();

  this.grow = function() {
    setBigger(10);
    setColor("red");
  };

  ...

  function getNextId() {
    var squares = document.querySelectorAll(".square");
    if (squares) {
      return squares.length;
    }
    return 0;
  }
}
```

This method is the *public interface* of the square object that is exposed to the rest of the code. You can use it to interact with a square object after it's been created. In order to make a square get bigger, call the `grow()` method. The details are handled by the square object.

The `grow()` method calls two other private functions: `setBigger()` and `setColor()`. Both these functions are nested in the constructor, so they are accessible to the `grow()` method, but not accessible to any code

outside of a square object. You can't call `square1.setBigger(10)` to make a square bigger; you must call `square1.grow()` instead.

**OBSERVE:**

```
function Square(size) {
  var initialSize = size;
  var el = null;
  var id = getNextId();

  this.grow = function() {
    setBigger(10);
    setColor("red");
  };

  ...

  function setBigger(growBy) {
    if (el) {
      size += growBy;
      el.style.width = size + "px";
      el.style.height = size + "px";
    }
  }

  function setColor(color) {
    if (el) {
      el.style.backgroundColor = color;
    }
  }

  ...

  function getNextId() {
    var squares = document.querySelectorAll(".square");
    if (squares) {
      return squares.length;
    }
    return 0;
  }
}
```

Both `setBigger()` and `setColor()` check to make sure that **the `el` variable has been initialized correctly** (which we'll do in the `display()` function), and then modify the style of the `<div>` element to grow the square and make sure it's got the right color, red.

Notice that we call two private functions from a public method. Even though the square objects created by the constructor have only one property, the method `grow()` (because of the closure we mentioned earlier) has access to the `setBigger()` and `setColor()` functions.

Also notice that the functions `setBigger()` and `setColor()` have access to the private variable, `el` (also because of the closure). So we can manipulate the value of a private variable by calling a public method. But we *can't* manipulate that private variable from outside the object. That gives you (as the creator of this `Square()` constructor, and the square objects that are made from it) greater control over how those objects are used.

## Accessing a Public Method from a Private Function

At this point, we've initialized our three private variables, and defined a public method, `grow()`. Now, we need to create the `<div>` element that will represent the square in the page.

To do that, we'll call the `display()` function. This is a nested function that first creates a new `<div>` element for the square, and then sets various properties to make the square the correct size and color.

We also set up the click handler for the `<div>` in this function. When you click on a `<div>` for a square, we want the square to grow, so we call the `grow()` method of the square object we're using. You might think we could write the `display()` function like this:

**OBSERVE:**

```
function display() {
  el = document.createElement("div");
  el.setAttribute("id", id);
  el.setAttribute("class", "square");
  el.style.width = size + "px";
  el.style.height = size + "px";
  el.innerHTML = "<p>" + id + "</p>";
  el.onclick = this.grow;
  document.getElementById("squares").appendChild(el);
}
```

However, when you call a nested function inside an object, **this** is set to the global window object, *not* the object the function is in. So, how do we refer to "this" object from inside the **display()** method?

We save the value of **this** in another variable, **self** (you can call this variable whatever you want, but by convention it's usually called **self** or **that**. It's a good idea to stick with this convention to make it easier for other programmers to understand your code). Once we've saved the value of **this** in **self**, we can call **display()**, and now **display()** can refer to the object's **grow()** method using **self.grow**:

**OBSERVE:**

```
function Square(size) {
  var initialSize = size;
  var el = null;
  var id = getNextId();

  this.grow = function() {
    setBigger(10);
    setColor("red");
  };

  var self = this;
  display();

  function setBigger(growBy) {
    if (el) {
      size += growBy;
      el.style.width = size + "px";
      el.style.height = size + "px";
    }
  }

  function setColor(color) {
    if (el) {
      el.style.backgroundColor = color;
    }
  }

  function display() {
    el = document.createElement("div");
    el.setAttribute("id", id);
    el.setAttribute("class", "square");
    el.style.width = size + "px";
    el.style.height = size + "px";
    el.innerHTML = "<p>" + id + "</p>";
    el.onclick = self.grow;
    document.getElementById("squares").appendChild(el);
  }

  function getNextId() {
    var squares = document.querySelectorAll(".square");
    if (squares) {
      return squares.length;
    }
    return 0;
  }
}
```

When you click on a square, the **grow()** method is called on the object stored in **self**, which is the square object that was created when you assigned the value of **self** to **this**. **self** is a private variable, and like the other private variables and methods, it is accessible internally to the square object, even though it's not accessible by the code using the square object.

Let's examine the contents of the square objects we create in the **window.onload** function by adding a couple of **console.log()**s to the code:

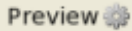
CODE TO TYPE: Update your JavaScript code in squaresAPI.html:

```
window.onload = function() {
  var square1 = new Square(100);
  var square2 = new Square(200);

  console.log(square1);
  console.log(square2);

  var growButton = document.getElementById("growButton");
  growButton.onclick = function() {
    square1.grow();
    square2.grow();
  };
};
```



and . In the console, the two square objects are displayed (Chrome browser):

```
▼ Square {grow: function} ⓘ
  ► grow: function () {
  ► __proto__: Square
▼ Square {grow: function} ⓘ
  ► grow: function () {
  ► __proto__: Square
```

You can see that the only property (that we added) in each square object is the **grow()** method.

Clicking on a square causes that square to grow, because we call the **grow()** method of the square on which you click.

The call to the square's **grow()** method is *set up* in the **display()** method, but the method isn't called until you actually click on a square. In order to allow you to see more explicitly that you can call the **grow()** method of a square using the **square1** and **square2** objects, we set up a form with one button, **Grow**, that calls the **grow()** method on both square objects:

OBSERVE:

```
window.onload = function() {
  var square1 = new Square(100);
  var square2 = new Square(200);

  var growButton = document.getElementById("growButton");
  growButton.onclick = function() {
    square1.grow();
    square2.grow();
  };
};
```

First, we **get the button element from the page**, and **add a click handler function to the button**. The click handler **calls the grow() method of both square1 and square2**. Because **grow()** is a public method, that is, a property of the object we are creating with the constructor, this method is accessible to code outside the object.

## Encapsulation and APIs

So, a square object has a lot of internal components that are not accessible to the "outside world." These are all the private variables and functions we've defined in the **Square()** constructor.

The one public method, **grow()**, that a square has is its *interface*: the point of interaction between the square and the rest of the code. The only method you're allowed to call from outside the square is **grow()**. We call this interface an *Application Programming Interface*, or API for short. When you see the term API used in software systems, it refers to the set of properties and methods that determine how software components should interact with each other.

We *encapsulated* all the private things we don't want code outside a square to be able to interact with: the `id`, the element representing the square, the initial size, and the functions used to create and manipulate the square. These private properties are not in the API; they're hidden and protected so they can't be used outside the square objects.

*Encapsulation* is a language mechanism for restricting access to an object's components. In some languages, like Java, we have keywords that specify which pieces of an object are private or public. In JavaScript, we don't have keywords to help us do this, but we can accomplish encapsulation by making variables and functions *private* to an object using the technique we've covered in this lesson. We sometimes refer to this technique as *information hiding*, because we're hiding the details of how an object is implemented to protect it from being used in the wrong way.

The advantage to encapsulation and providing an API to an object like a square is that it prevents code that is using the square from setting the internal state of the object to an invalid state. We are unable to modify the `id` of the square, or the element that represents the square in the page—both of which would disrupt how the square works—because those values are now protected from outside manipulation.

In JavaScript, the mechanism we use to encapsulate data is not perfect. For instance, you could easily change the `grow()` method of a square by writing:

OBSERVE:

```
square1.grow = 3;
```

...and break your squares! Yet encapsulation lets us hide most of the details of how a square works, so that in order to use a square, we only have to know one thing about it: to grow it, we call the `grow()` method. Everything else is handled for us. Hopefully you'll know better than to change the public interface of an object by setting the properties to something else.

Let's review. We defined a `Square()` constructor that creates new square objects with one public method, `grow()`, and various private variables (sometimes called private "members") and methods. Any properties and methods, for instance `grow()`, that are added to the object at construction time (either through the constructor, or through the object's prototype) will be accessible to code that uses the object. Any variables and functions that are defined in the constructor, like `id` and `display()`, will not be accessible to code outside the object. The interface (API) for our square objects is the `grow()` method: this is the only method that code outside the square objects can call.

Defining an interface for your objects is particularly important if you're writing code that you're sharing with others. Perhaps you're working on components for a product at work and other members of your developer group will be using your components. Or perhaps you're developing a library, like jQuery, that you want to make available online. By encapsulating the details of how an object works and providing a simple interface for using the object, you'll be making your objects easier to understand and easier to use.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Closures

## Lesson Objectives

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

- create and use a closure to "remember" values.
- explain how a closure is created.
- use a closure to store a value for a click handler.

## Closures

In the previous lesson, you learned about encapsulation and information hiding. All of that functionality is possible because of a feature of JavaScript: *closures*. Closures appear to be relatively straightforward, yet they can be really difficult to wrap your head around. This entire lesson is devoted to closures: what they are, how they work, and when to use them.

### Making a Closure

We've mentioned closures before, now it's time to uncover the mystery that surrounds them. We'll start with some basic examples and then come back to a couple of examples from the previous lessons to see how we've used closures in the past.

Create a new file add this code:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>


    function makeAdder(x, y) {
      var adder = function() {
        return x + y;
      };

      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **closure.html**, and . Open the console (reload the file if necessary) and you'll see, "Result is: 5."

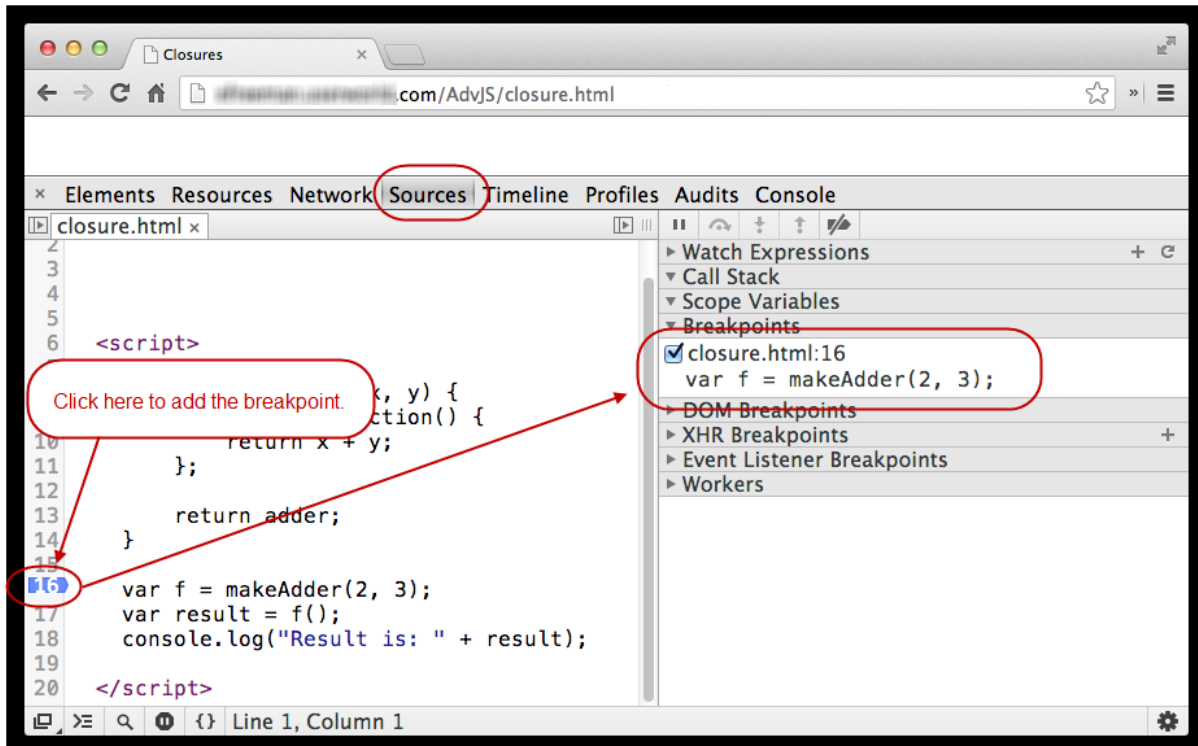
In this code we've got a function that returns another function. You may remember from an [earlier lesson](#) that functions are *first-class values*; that is, you can return a function from a function, just like you can return values like 3 or "string." The function **makeAdder()** returns a function, **adder()**, that adds two values and returns the result. We've named the function **adder()** inside **makeAdder()**, but when we return it from **makeAdder()**, we name it **f()**. So, in order to call the returned function, we use **f()**. (We could use the same name for the function in both places, but for this lesson we want to have a way to identify the function when it's *defined*, and the function when it's *called*.)

The two values that we want to add together are passed to the **makeAdder()** function as arguments when we call it. The function **adder()** doesn't take any arguments. Instead, **adder()** adds together the two numbers passed into **makeAdder()**. So, how does this work?

Local variables (including parameters of functions) disappear after the function within which they are defined is done executing. So, while the parameters **x** and **y** are defined when the function **adder()** is *created*, when we *call* it later, using the name **f()**, **x** and **y** are long gone. So how does calling **f()** return the (correct) value, 5? It seems like **f()** somehow "remembers" the values of **x** and **y** which were defined when **f()** was created (as **adder()**).

Well, that's exactly what happens. The function **f()** "remembers" the values of **x** and **y** through a closure. Let's step through the execution of this code to see the closure in the JavaScript console.

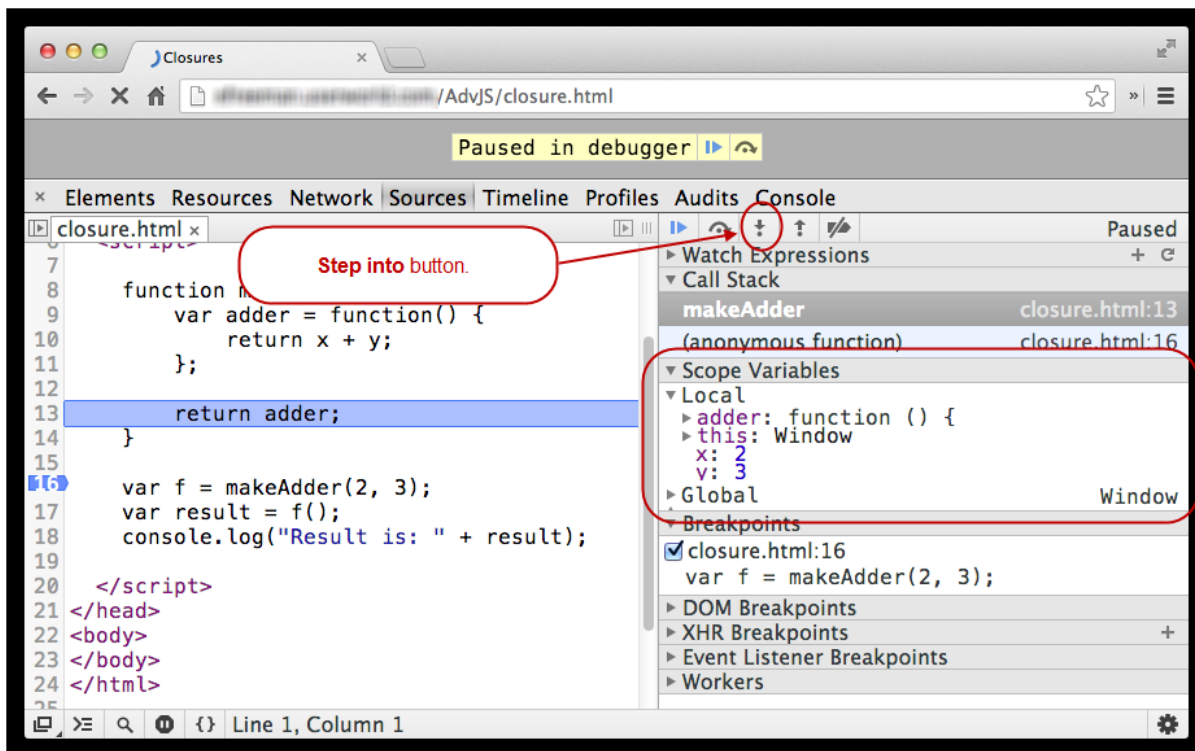
First, open the file **closure.html** in the Chrome browser, and add a breakpoint to the line of code where we call **makeAdder()**:



Add a breakpoint, click the **Sources** tab, and then open **closure.html** (from the left pane). Click on the line number next to the line of code where you want to add the breakpoint (in our version, that's line number 16). The breakpoint appears under the **Breakpoints** section in the right pane.

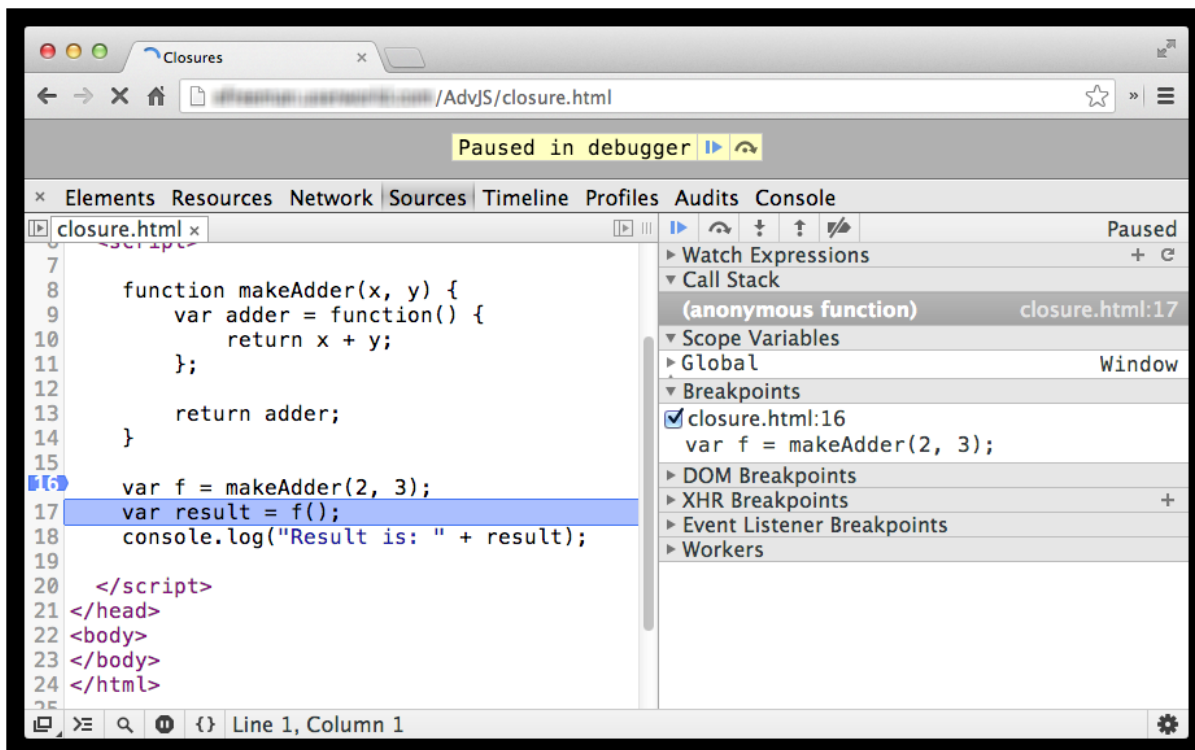
Now, reload the page and click the **Step into** button twice. This is the third button from the left of the right pane, with a little arrow pointing down on top of a period:



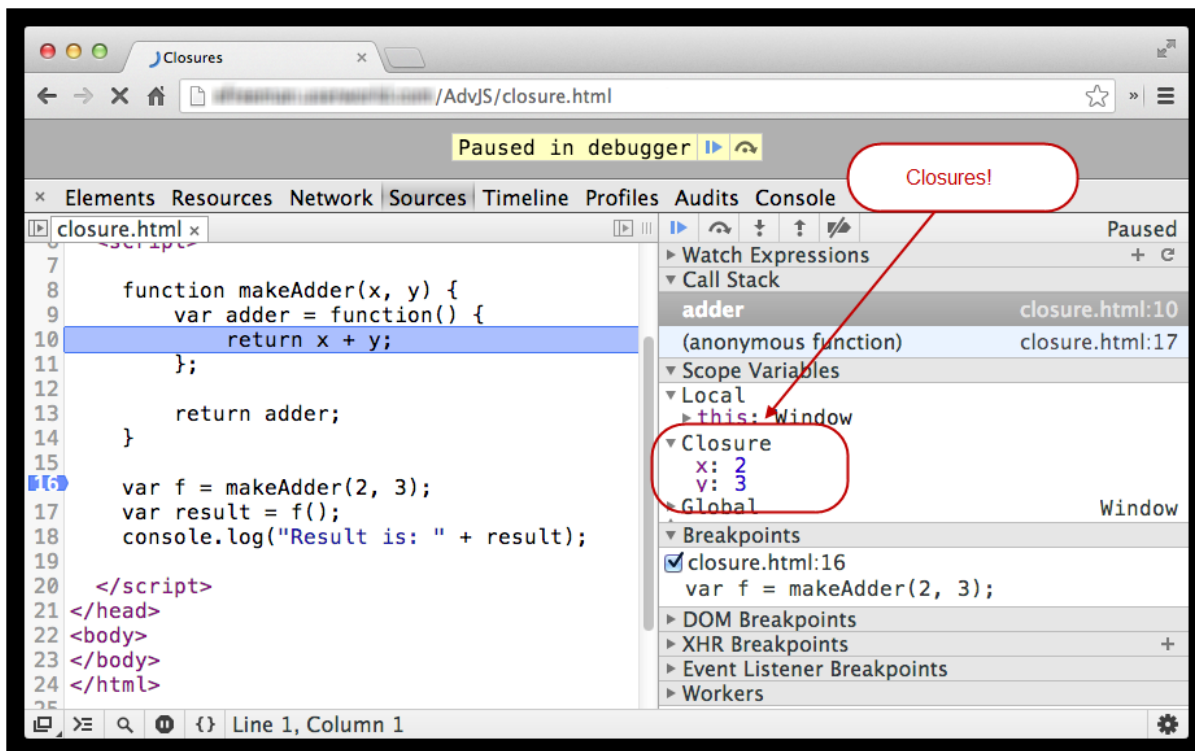


We're now on the line where we return `adder()`. Look in the right pane under **Scope Variables**. The local variables that are defined inside `makeAdder()`, including the two parameters, `x` and `y`, as well as the `adder()` function. These are all local variables because they are defined within `makeAdder()`.

Click **Step into** twice more; we're now stopped on the line where we call `f()` (we haven't executed this line yet). `f` is a global variable (if you open **Global** under Scope Variables in the right pane, you'll see it defined as a global variable, which is the global window object).

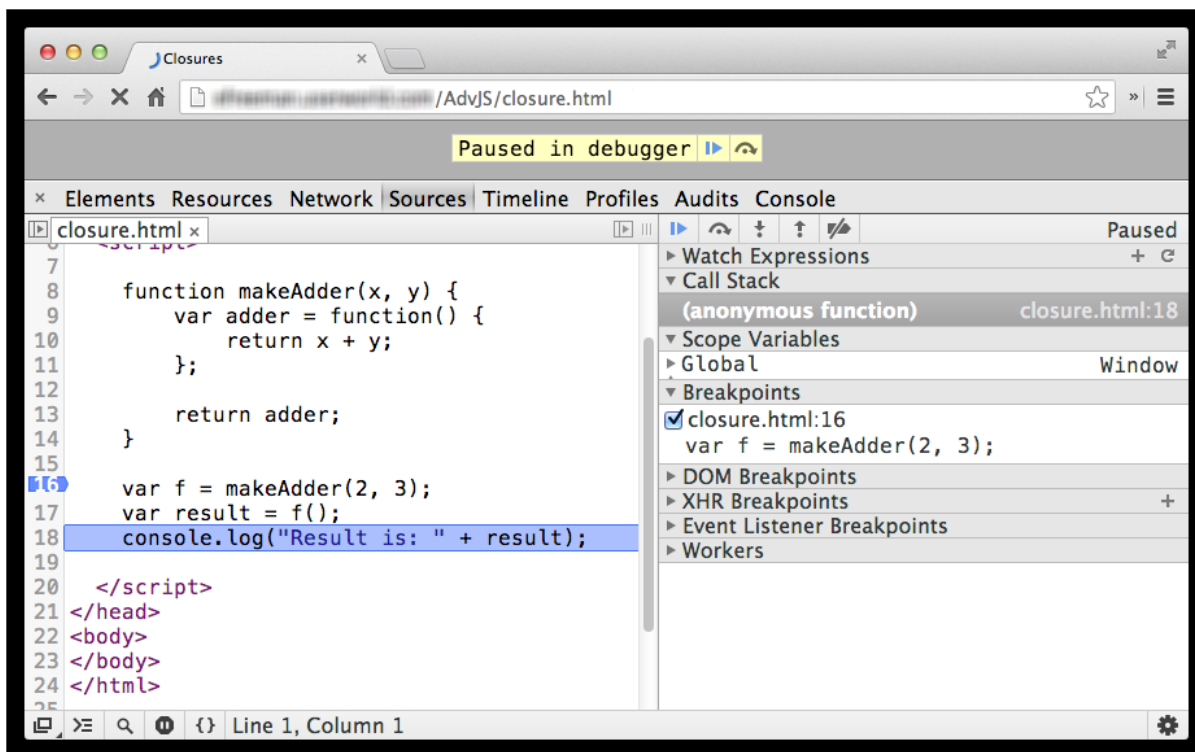


Click **Step into** once more, so that we call `f()` and stop just before we return the result of adding `x` and `y`:



Under Scope Variables, in Local, you see something called **Closure**. Inside that, you'll see two variables: **x** and **y**, with their values set correctly to 2 and 3—the arguments we passed into **makeAdder()** earlier. This is the closure, where **f()** gets its two values to add together.

**Step into** twice again to execute the line of code, **return x + y**, and return from **f()**. The closure disappears:



Click **Resume script execution** to complete the script execution.

## What is a Closure?

Now you've *made* a closure, but what exactly *is* a closure?

To understand a closure, you need to remember how *scope* works. In the earlier lesson on scope, we talked

about how the *scope chain* is created when you call functions. The scope chain is a series of scope objects containing the values of the variables in a function's scope. For global functions (that is, functions defined at the top level), we have just two scope levels: the local scope (the scope within the function) and the global scope. When you call a function and refer to a variable, we get the value for that variable first by looking in the local scope and, if we can't find it there, we look in the global scope.

Now, recall that when we call a nested (or "inner") function, we have three scope levels: the scope of the nested function, then the scope of the function containing the nested function, and finally the global scope.

In our example, we created a nested function named `adder()`:

```
OBSERVE:
function makeAdder(x, y) {
  var adder = function() {
    return x + y;
  };

  return adder;
}

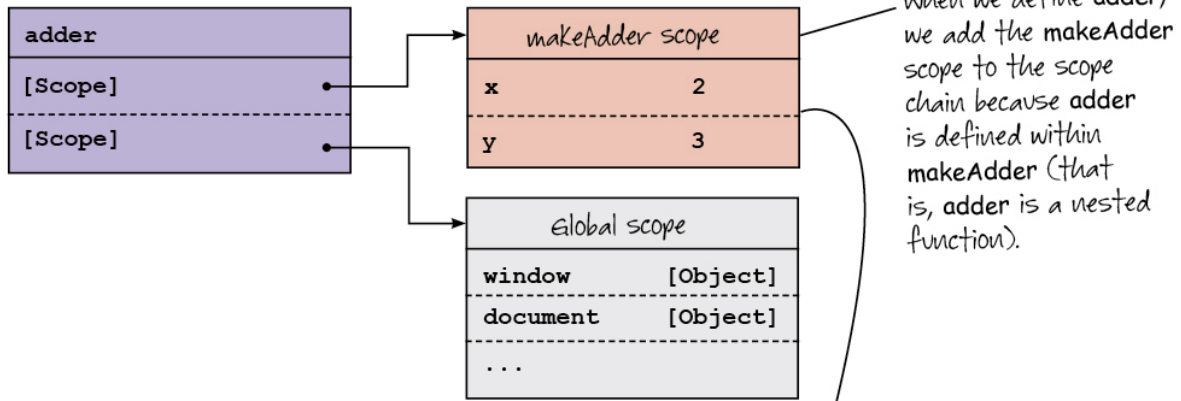
var f = makeAdder(2, 3);
var result = f();
console.log("Result is: " + result);
```

Inside `adder()`, we refer to two variables, `x` and `y`. These variables are defined in the `makeAdder()` function, so they are *not* local to `adder()`. If we just called `adder()` from inside `makeAdder()` (for example, if we changed the line `return adder` to `adder()`), you'd see that in order to figure out the values of the variables `x` and `y`, we'd use the scope chain. We'd look for those values in the scope of `adder()` first, but we wouldn't find them there so we'd look for those values in the scope of `makeAdder()`, and we'd find them there.

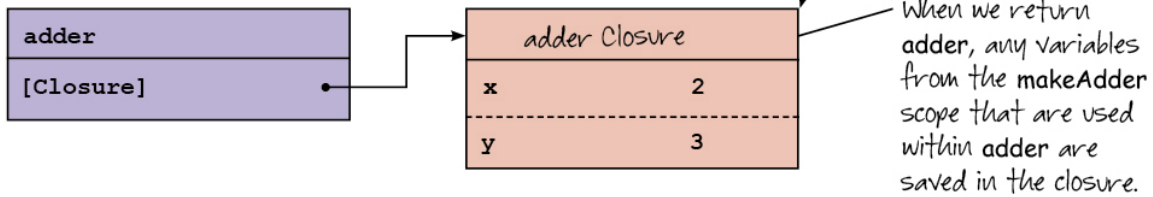
However, we're not calling `adder()` from inside `makeAdder()`; we're *returning* `adder()` from `makeAdder()`. When we return the `adder()` function, it comes along with a scope object: an object that contains the variables that are in the scope of `adder()` when `adder()` is created. That includes the two variables `x` and `y`. This object is essentially the same as the *scope object* of `makeAdder()` that was created for the scope chain. It's the *context* within which `adder()` is created.

This object is the closure. A closure is an object that captures the context in place when a function is created. The closure "remembers" all the variables that are in scope at the time the inner function is created. If we just call the inner function right away, the closure gets thrown away when the containing function ends, but if we *return* that inner function, the closure comes along with it:

When we define the `adder` function:

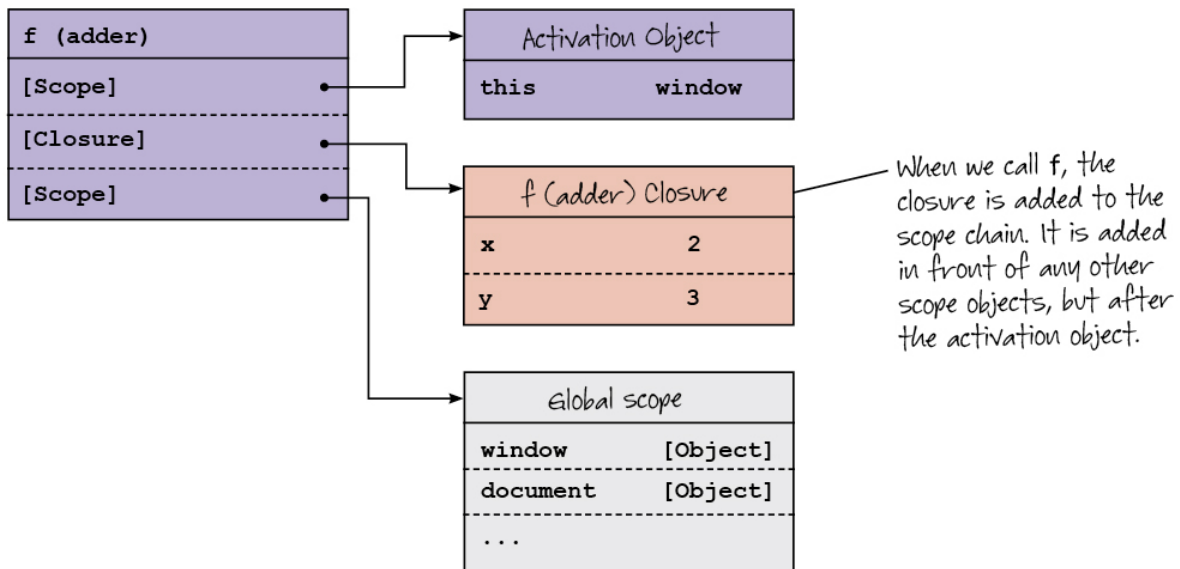


When we return the `adder` function:



When we call that function later and look for the values of the variables it refers to, if we don't find those values in the function itself (that is, they aren't local variables), we look in the closure:

When we execute the `f` function:



So the closure becomes part of the scope chain when you call a function.

The closure looks a lot like the scope object that's added to the chain when we call a nested function from its containing function (for example, if we called `adder()` from inside `makeAdder()`). Although here, we use the closure instead of the scope object because we're calling the returned function (`f()`) *after* the containing function has returned, so we can't use the normal scope chain to find the values of the variables in `f()`; we have to use the closure instead. We've "captured" the scope, or context, within which the nested function was created in the closure so we can refer to the variables long after the containing function has completed execution.

## Playing with Closures

Let's play with closures a bit so you can see how they work. First, let's prove that only variables that are actually referenced by an inner function are added to a closure. Modify `closures.html` as shown:

```
CODE TO TYPE:



<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>

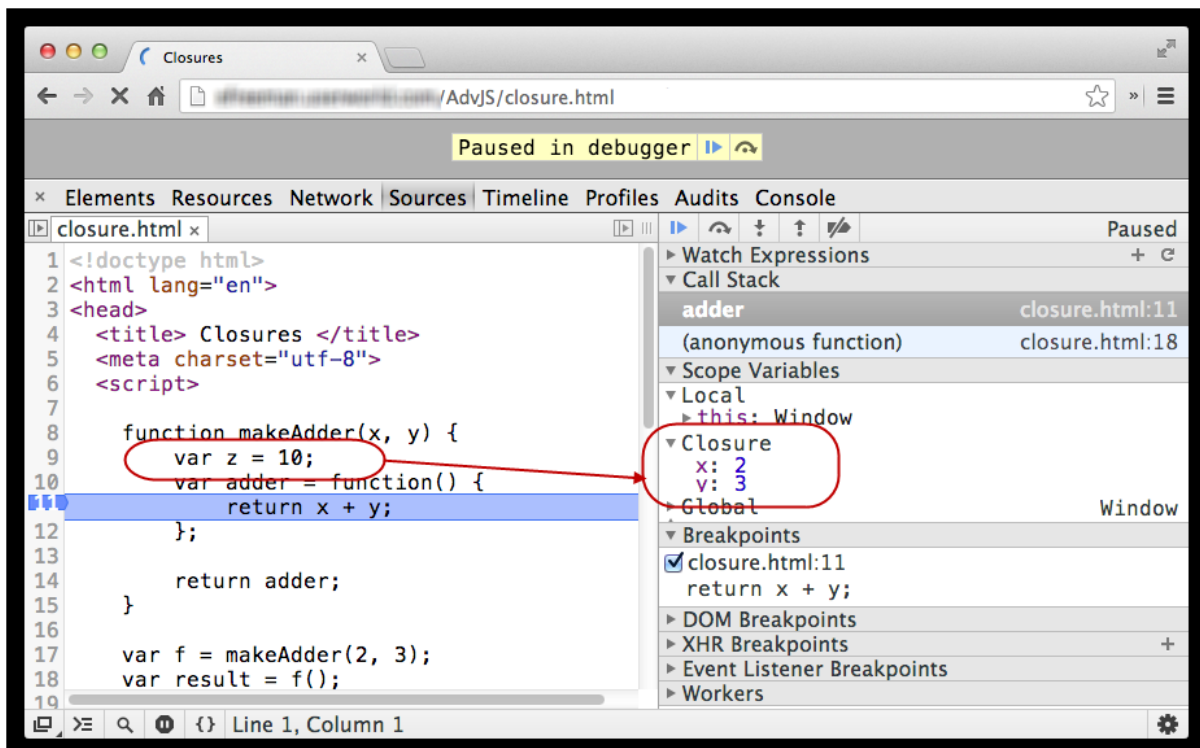
    function makeAdder(x, y) {
      var z = 10;
      var adder = function() {
        return x + y;
      };

      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

  </script>
</head>
<body>
</body>
</html>
```

 and . In the console, open the **Sources** tab, and look at `closure.html`. Clear the previous breakpoint and add a breakpoint to the line where we return the result of adding `x` and `y` (inside `adder()`), like this:



Reload the page. The execution will stop at the breakpoint, so you can inspect the closure. Notice that even though we've added a local variable `z` to `makeAdder()`, that variable is not included in the closure. Why?

Because it's not referenced by `adder()`, so it's not needed in the closure.

Next, let's prove that only non-local variables are added to a closure:

```
CODE TO TYPE:



<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>

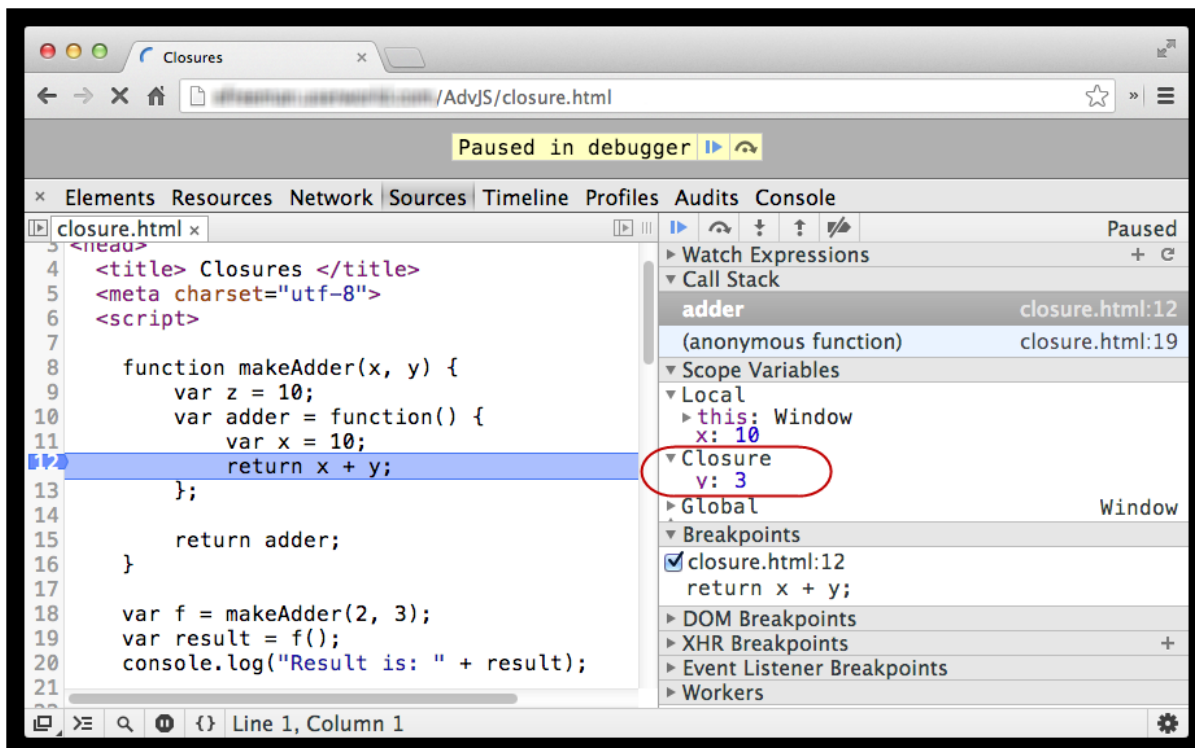
    function makeAdder(x, y) {
      var z = 10;
      var adder = function() {
        var x = 10;
        return x + y;
      };

      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

  </script>
</head>
<body>
</body>
</html>
```

 and  preview. In the console, open the **Sources** tab, and look at `closure.html`:



Now, only `y` is in the closure. The local variable `x` shadows the parameter `x` in `makeAdder()`, so `x` is no longer needed in the closure—we'll always use the value of the local variable if we refer to `x` in `adder()`. Also, look at the console (click on the **Console** tab); the result is now 13 instead of 5.

What do you think will happen if we refer to **z** in **adder()**?

#### CODE TO TYPE:

```

<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>



    function makeAdder(x, y) {
      var z = 10;
      var adder = function() {
        var x = 10;
        return x + y + z;
      };

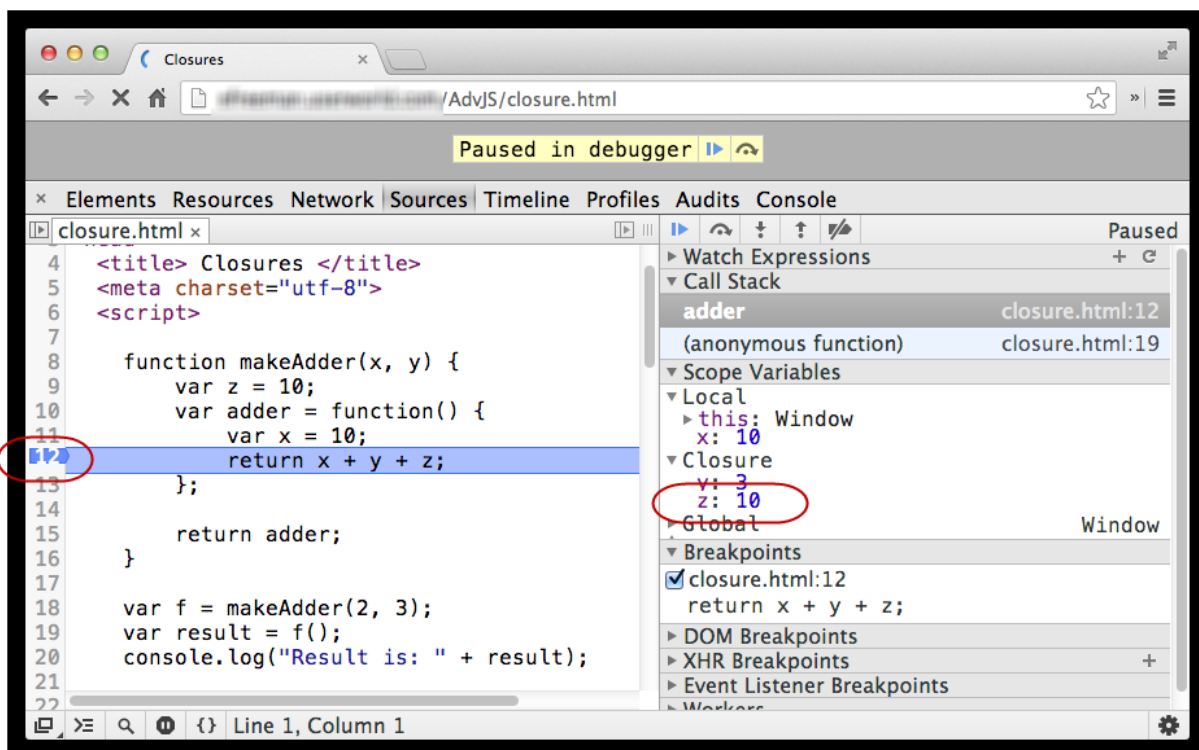
      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

  </script>
</head>
<body>
</body>
</html>

```

 and . In the console, open the **Sources** tab, and look at **closure.html**. Notice we're now adding **z** to **x** and **y**. Add a breakpoint to the line where we return the result of adding **x**, **y** and **z** (inside **adder()**), like this:



Reload the page. When the execution stops, you'll see that **z** is now included in the closure, because it's referenced by **adder()**.

What do you think will happen if you remove the declaration of **z**, but leave the reference to **z** in **adder()**? Try it

and see!

## Each Closure is Unique

Each time you call **makeAdder()**, you'll get a function back that adds two numbers together. Let's make another function, **g()**, that adds together the numbers 4 and 5:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>

    function makeAdder(x, y) {
      var z = 10;
      var adder = function() {
        var x = 10;
        return x + y + z;
      };

      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

    var g = makeAdder(4, 5);
    var anotherResult = g();
    console.log("Another result is: " + anotherResult);

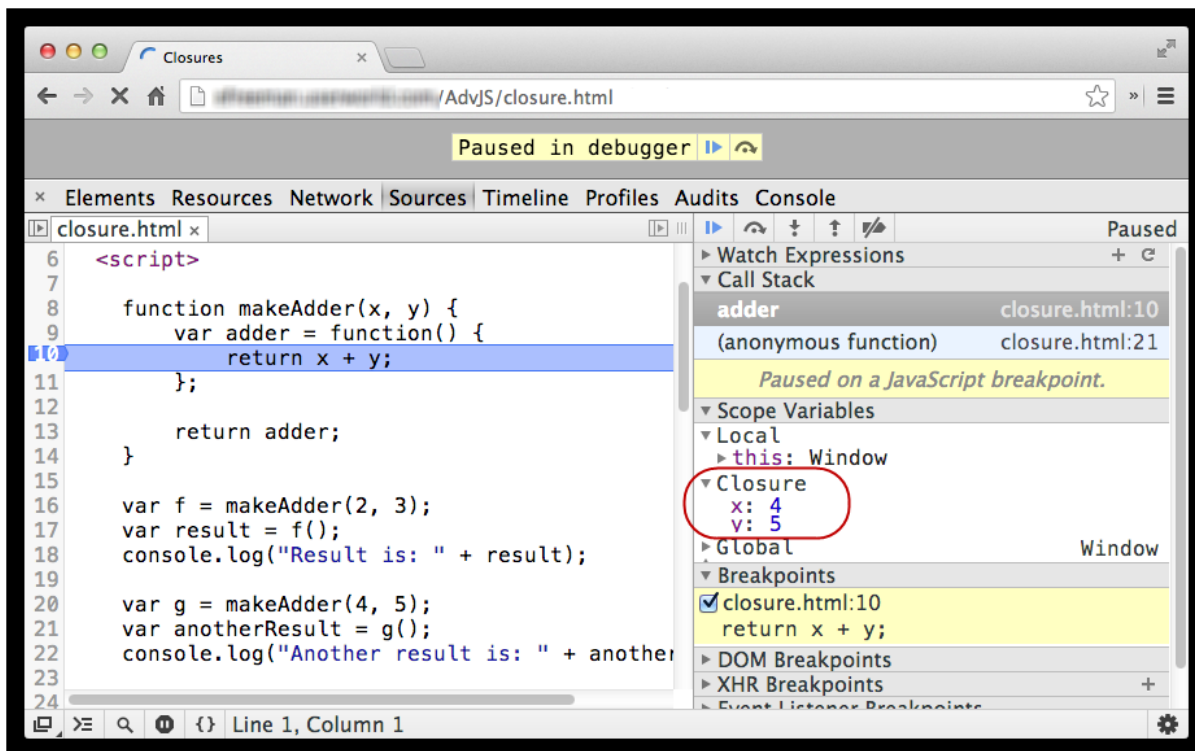
  </script>
</head>
<body>
</body>
</html>
```



and **Preview**. In the console, you'll see that the result of calling **g()** ("Another result") is 9. Now, open the **Sources** tab, and look at **closure.html**. Add a breakpoint at the line where we return the result from **adder()** (line 10 in our version), and reload the page. The first time the execution stops at the breakpoint, we're calling **f()**, so you'll see the values 2 and 3 for **x** and **y** in the closure. Click **Resume script execution** again. Now when execution stops at the breakpoint, we call **g()**, so you'll see the values 4 and 5 for **x** and **y** in the closure.

In other words, **f()** and **g()** get separate closures containing different values for **x** and **y**. You can call **f()** again after calling **g()**, and you'll still get the right answer. When **f()** and **g()** are created, the values of **x** and **y** are different, so each function has a separate closure, each with different values for the variables that are in it. Remember, a closure captures the context of a function when that function is *created*.





## Closures Might Not Always Act Like You Expect

Now, try this:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>

    function makeAdder(x, y) {
      var adder = function() {
        return x + y;
      };
      x = 10;
      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

    var g = makeAdder(4, 5);
    var anotherResult = g();
    console.log("Another result is: " + anotherResult);

  </script>
</head>
<body>
</body>
</html>
```



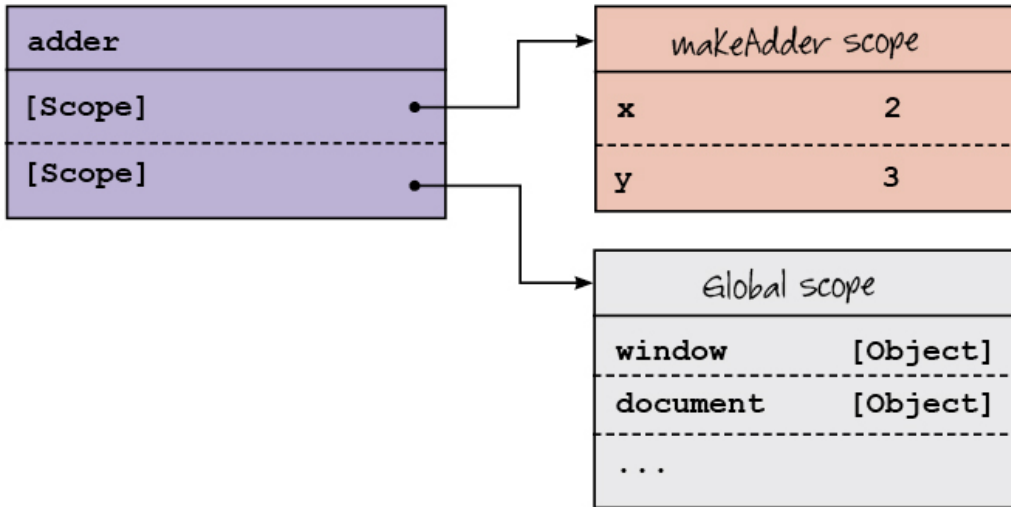
and



Look at the result in the console. The result of calling **f()** is now 13, and the result of calling **g()** is now 15. If you still have the breakpoint at line 10, you'll see that the closure now contains the value 10 for **x** in both functions. That's because a closure is a *reference to an object*: an object that contains

the values of the variables in the scope of the function associated with the closure. So, when we call **makeAdder()** the first time, we create the function **adder()** by defining it in **makeAdder()**. The closure is created, and **adder()** gets a reference to that closure object, which contains a property named **x** with the value 2 (the value we passed into **makeAdder()**). Before we return the **adder()** function value, we change the value of the property in the closure object associated with **adder()**. **adder()** still points to the same closure object, but we've *changed* the value in that closure object, so later, when we call **f()** (which is just another name for the **adder()** function we created when we called **makeAdder(2, 3)**), we look up the value of **x** and find 10 instead of 2:

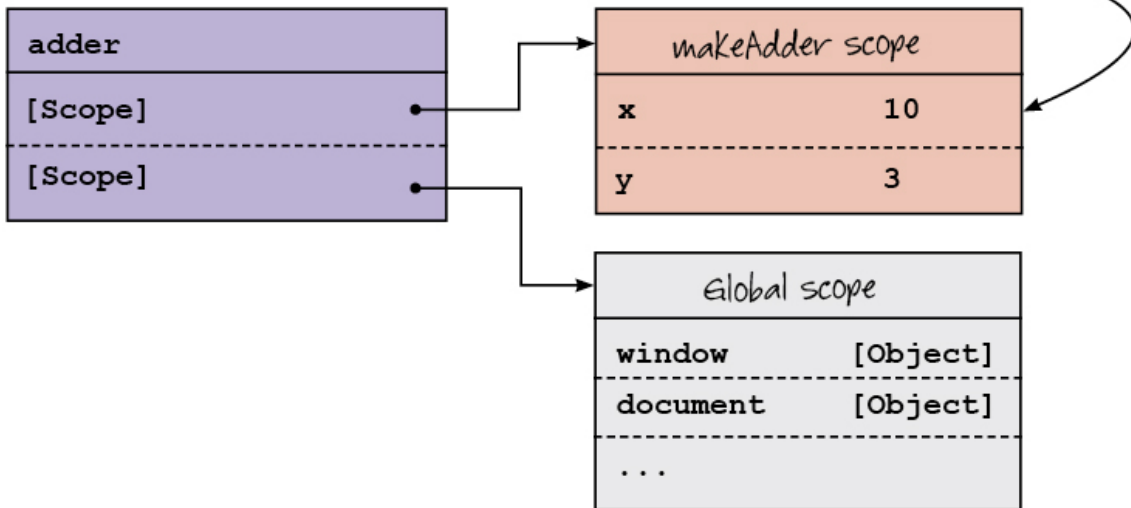
We define the adder function:



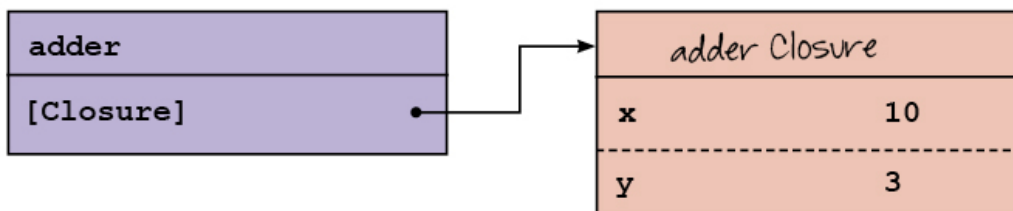
Then we change the value of x:

`x = 10;`

Which changes the value in the scope object:



So when we return the adder function, the value of x in the closure is 10:



This is really important to understand, so look it over a couple of times. If you don't remember that the closure

associated with a function is an object, and so what's stored in the function object as the closure is actually a *reference to an object*, you could run into problems. If you change that object *after* you've created the function, that function will use the *new* values in the object, not the original ones.

## Closures for Methods

Closures work for methods (which are just functions in objects) too:

```
CODE TO TYPE:

<!doctype html>
<html>
<head>
  <title> Closures </title>
  <meta charset="utf-8">
  <script>

    function makeAdder(x, y) {
      var adder = function() {
        return x + y;
      };
      x = 10;
      return adder;
    }

    var f = makeAdder(2, 3);
    var result = f();
    console.log("Result is: " + result);

    var g = makeAdder(4, 5);
    var anotherResult = g();
    console.log("Another result is: " + anotherResult);

    function makeObject(x, y) {
      return {
        z: 10,
        adder: function() {
          return x + y + this.z;
        }
      };
    }

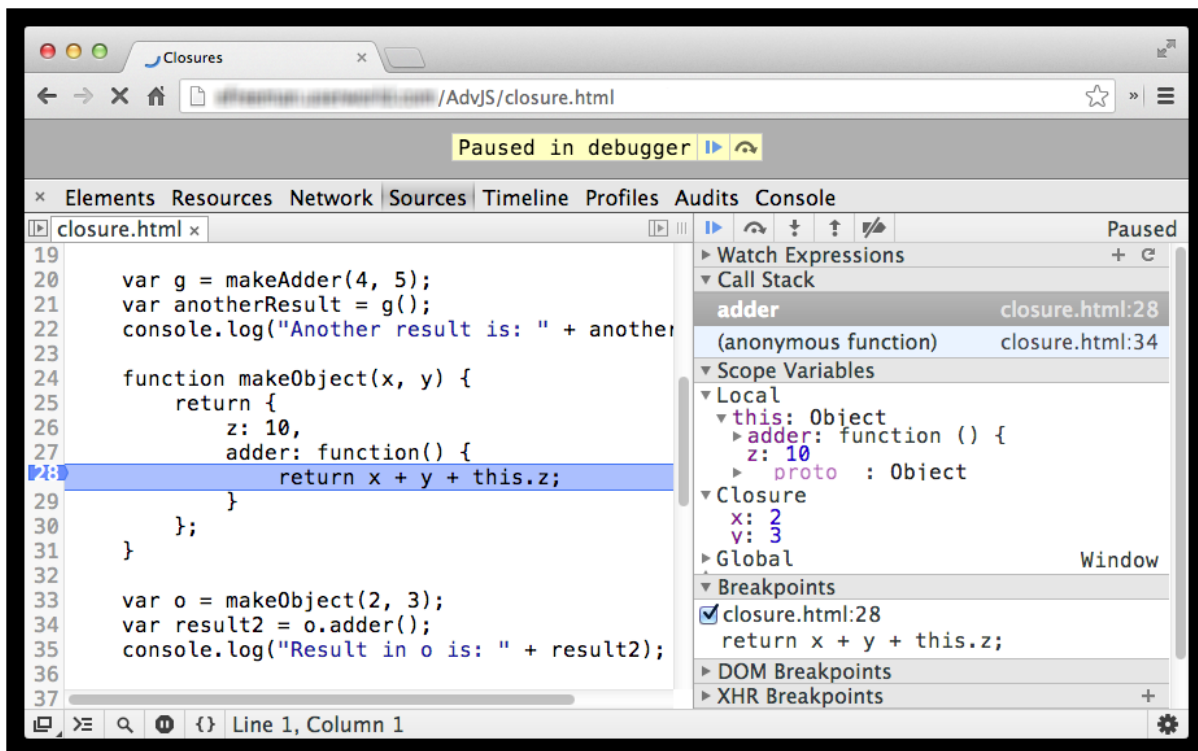
    var o = makeObject(2, 3);
    var result2 = o.adder();
    console.log("Result in o is: " + result2);

  </script>
</head>
<body>
</body>
</html>
```



and **Preview** . In the console, you see that the value of **result2** is 15.

Go ahead and experiment by setting a breakpoint at the return in the new **makeObject()** function (line 28 in our version). When you execute the code with the breakpoint in place, you'll see that when we call **o.adder()** the value of **this** is the object **o** (good!) and that object contains two properties: the method **adder()** and the property **z** that has a value of 10. The closure contains the values 2 and 3 for **x** and **y**. We don't need the value of **this.z** in the closure, because we get that value from the object that contains method we're calling—that is, **o**.



## Using Closures

Now you know how closures work, so when are they useful. After all, it's not often that we create functions that return other functions in our everyday code. Let's look at a few examples of where closures come in handy.

### Using Closures to Create Private Data

We'll start by creating a function that counts. Create a new file and add this code:

#### CODE TO TYPE:


```
<!doctype html>
<html>
<head>
  <title> Counter </title>
  <meta charset="utf-8">
  <script>

    function makeCounter() {
      var count = 0;
      return function() {
        count = count + 1;
        return count;
      };
    }

    var count = makeCounter();
    console.log("Counter: " + count());
    console.log("Counter: " + count());
    console.log("Counter: " + count());
    console.log("Counter: " + count());
    console.log("Counter: " + count());

  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **counter.html**, and **Preview** . Open the console (and reload the file if necessary) and you see:

OBSERVE:
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5


This is kind of cool because we've used a closure to encapsulate the counter variable, and the process of counting. In other words, the counter variable, **count**, is totally private, and the only way to increment it is to call the function **count()**. The **count** variable exists only within the closure for the function **count()**, so no one can come along and change the value of the counter by doing anything other than calling **count()**.

### Closures as Click Handlers

So far we've looked at examples that return a function from a function, and seen how the function that is returned comes along with a closure object. Another way to use a function after the context within which the function is created has gone away is to assign a function to an object property; for instance, like when we assign handlers for events to a property in an object, like a `<div>`. Create a new file to look at a common use of closures—and a common mistake that goes along with it:

CODE TO TYPE:
<pre> &lt;!doctype html&gt; &lt;html&gt; &lt;head&gt;   &lt;title&gt; Closures for divs &lt;/title&gt;   &lt;meta charset="utf-8"&gt;   &lt;style&gt;     div {       position: relative;       margin: 10px;       background-color: red;       border: 1px solid black;       width: 100px;       height: 100px;     }   &lt;/style&gt;   &lt;script&gt;     window.onload = function() {       var numDivs = 3;       for (var i = 0; i &lt; numDivs; i++) {         var div = document.getElementById("div" + i);         div.onclick = function() {           console.log("You just clicked on div number " + i);         };       }     };   &lt;/script&gt; &lt;/head&gt; &lt;body&gt;   &lt;div id="div0"&gt;&lt;/div&gt;   &lt;div id="div1"&gt;&lt;/div&gt;   &lt;div id="div2"&gt;&lt;/div&gt; &lt;/body&gt; &lt;/html&gt; </pre>



Save this in your **/AdvJS** folder as **divsClosure.html** and **Preview** . You see three red squares. Open the console and try clicking on each of the squares. Each time you do, you'll see "You just clicked on div number 3." Can you figure out why we get "3" each time, instead of the correct value for each `<div>` (that is, 0, 1 or 2)? Think about it for a few minutes before you go on.

First, we've got three different <div>s in the page, with the ids "div0," "div1," and "div2." We want to add a click handler to each <div>. Each click handler will do the same thing: display a message showing the number corresponding to the number in the id of the <div>. So if you click on the "div0" <div>, you'll see the message, "You just clicked on div number 0," and likewise for <div>s 1 and 2.

So, in the code, we create a for loop to iterate through the three <div>s and add a click handler to each. We use the loop variable *i* for the number of the <div> so we can display that number in the click handler. Notice that the **click handler function** that we assign to each of the <div> objects references the variable *i*, and *i* is not defined in the **click handler function**—it's defined in the scope surrounding the **click handler function** (that is, in the **window.onload** function).

So what happens? A closure is created! When we store the function value of the **click handler function** in the **onclick** property, that function comes along with a closure that contains the variable *i*. Later, when you click on the <div>, the value of *i* will be found in the closure associated with the **click handler function**.

```
OBSERVE:

window.onload = function() {
  var numDivs = 3;
  for (var i = 0; i < numDivs; i++) {
    var div = document.getElementById("div" + i);
    div.onclick = function() {
      console.log("You just clicked on div number " + i);
    };
  }
};
```

That all sounds good, but it's not working right. We see 3 every time we click on any of the <div>s instead of the correct number for the <div>. Why?

Well, remember, a closure associated with a function is an object, and the function contains a *reference* to that closure object. So if we change the value of a variable that's captured in the closure *after* we create that closure, we're changing the value of the variable that will be used when we call that function later.

In this example, each time we set the **click handler function** to the **onclick** property of a <div>, the value of *i* will be correct initially, but then we change the value of *i* the next time through the loop, which changes the value in the closure we just made.

Our loop stops iterating when the value of *i* is 3. So when we call any of those click handler functions later (like when you click on a <div>), you see the value of *i* that was in place at the end of the loop, *not* the value of *i* that was in place when the closure was created originally.

Try using the console to add a breakpoint in the code to inspect the closure. Add the breakpoint on the line in the click handler function where we use **console.log()** to display the <div> information. Click on a <div> to see the closure when the click handler function is called.

We can fix this by creating another closure. Let's see how:

```
CODE TO TYPE:

...
window.onload = function() {
  var numDivs = 3;
  for (var i = 0; i < numDivs; i++) {
    var div = document.getElementById("div" + i);
    div.onclick = function() {
      console.log("You just clicked on div number " + i);
    };
    div.onclick = (function(divNum) {
      return function() {
        console.log("You just clicked on div number " + divNum);
      };
    })(i);
  }
};
```



and **Preview**. Try clicking on each <div> again. Now you get the correct number values for each <div>.

How does this work?

OBSERVE:

```
window.onload = function() {
  var numDivs = 3;
  for (var i = 0; i < numDivs; i++) {
    var div = document.getElementById("div" + i);
    div.onclick = (function(divNum) {
      return function() {
        console.log("You just clicked on div number " + divNum);
      };
    })(i);
  }
};
```

When we assign the value of each <div>'s **click handler function**, we do so by executing **another function** that returns a function value for the **click handler**. **This function** executes right away. It seems a little odd because we're putting the function expression in parentheses first, and after the function expression, we have another set of parentheses:

OBSERVE:

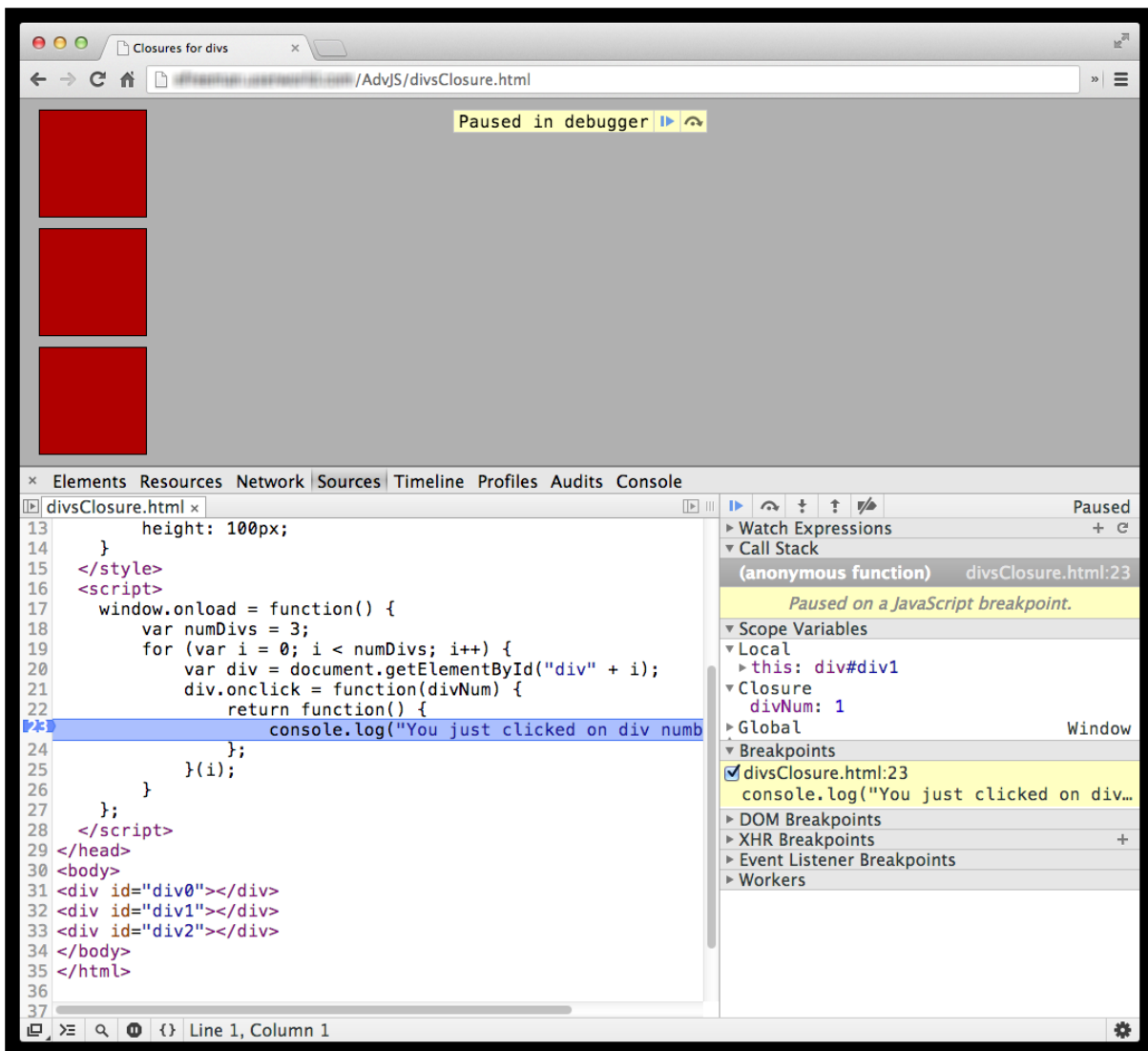
```
div.onclick = (function(divNum) { ... }) (i);
```

We're *calling* the **function we just created**. (We'll talk more about this pattern of creating and calling a function in one step in a later lesson).

Putting the function expression in parentheses makes sure the function expression is treated as an expression, and not a function declaration. Also, we pass the value of *i* into the **function we're calling**. The value gets passed into a variable **divNum**, which is used by **a function** we're returning from the **function we just called**. When we return **a function** from **a function**, we create a closure that contains any variables referenced by the **the function being returned** that are defined in the **function that contains it**. In this case, both of these functions are anonymous; we're not actually giving them names like we did before with **makeAdder()** and **adder()**, but that's okay. The closure works in exactly the same way. In this case, the closure associated with the **the function being returned** contains the value of **divNum**. Note that this value *does not change*. Even if the value of *i* changes, the value of **divNum** in the closure does not (remember that arguments are passed by value to functions, so **divNum** gets a *copy* of the value in *i*).

The **function that's returned** is assigned to the **div.onclick** property, so it is available once the **window.onload** function has completed. That means that the values in that function's closure are also available, so when you click on a <div>, you'll get the correct number for that <div> because you're accessing the **divNum** variable in the closure. Add a breakpoint to the code on the **console.log()** line again (line 23 in our version), and inspect the closure when you click on a <div>. You'll see the variable **divNum** and the correct number for the <div> you clicked on:





Using a closure like this to capture the current value of a variable by passing it to a function that returns another function is a common technique used by JavaScript programmers (and one we'll look at more in the next lesson).

## Where We've Used Closures Before

Before we end the lesson, let's look at two examples from earlier in the course where we used closures.

First, take another look at the example, [AdvJS/functions3.html](#) from the [Functions lesson](#). (If you don't have this file, no worries—you can copy it in and save it as [AdvJS/functions3.html](#).)

CODE TO TYPE: This code is in the file functions3.html in your AdvJS/ folder

```
<!doctype html>
<html>
<head>
  <title> Returning Functions </title>
  <meta charset="utf-8">
  <script>
    function makeConverterFunction(multiplier, term) {
      return function(input) {
        var convertedValue = input * multiplier;
        convertedValue = convertedValue.toFixed(2);
        return convertedValue + " " + term;
      };
    }

    var kilometersToMiles = makeConverterFunction(0.6214, "miles");
    console.log("10 km is " + kilometersToMiles(10));

    var milesToKilometers = makeConverterFunction(1.62, "km");
    console.log("10 miles is " + milesToKilometers(10));
  </script>
</head>
<body>
</body>
</html>
```

We created this example to show how to return a function from a function. The function we return from **makeConverterFunction** references the two parameters: **multiplier** and **term**. When we call the returned function later (as **kilometersToMiles()** or as **milesToKilometers()**), we'll use the closures associated with the two functions that captured the context—the values of the parameters when the function was defined and returned—to determine the values of those variables.

Try adding a breakpoint inside the function that's returned (within **makeConverterFunction**) so you can see the closures in action.

Let's also look again at the **squaresAPI.html** example from the [Encapsulation and APIs lesson](#). In that lesson we talked about encapsulation and information hiding. We used a constructor, **Square()**, to create objects, but kept some of the data in the object being created private by not assigning values to properties of the object. Instead we used local variables and nested functions. (Again, if you no longer have the file **squaresAPI.html** in your **AdvJS/** folder, feel free to copy it in from here and save it as **AdvJS/squares.html**.)

```
<!doctype html>
<html>
<head>
  <title> Squares with API </title>
  <meta charset="utf-8">
  <style>
    .square {
      background-color: lightblue;
      cursor: pointer;
    }
    .square p {
      padding-top: 35%;
      text-align: center;
      -webkit-user-select: none;
      -moz-user-select: none;
      -ms-user-select: none;
      user-select: none;
    }
  </style>
  <script>
    function Square(size) {
      var initialSize = size;
      var el = null;
      var id = getNextId();

      this.grow = function() {
        setBigger(10);
        setColor("red");
      };

      this.reset = function() {
        setBigger(initialSize - size);
        setColor("lightblue");
      };

      var self = this;
      display();

      function setBigger(growBy) {
        if (el) {
          size += growBy;
          el.style.width = size + "px";
          el.style.height = size + "px";
        }
      }

      function setColor(color) {
        if (el) {
          el.style.backgroundColor = color;
        }
      }

      function display() {
        el = document.createElement("div");
        el.setAttribute("id", id);
        el.setAttribute("class", "square");
        el.style.width = size + "px";
        el.style.height = size + "px";
        el.innerHTML = "<p>" + id + "</p>";
        el.onclick = self.grow;
        document.getElementById("squares").appendChild(el);
      }

      function getNextId() {
        var squares = document.querySelectorAll(".square");
        if (squares) {
```

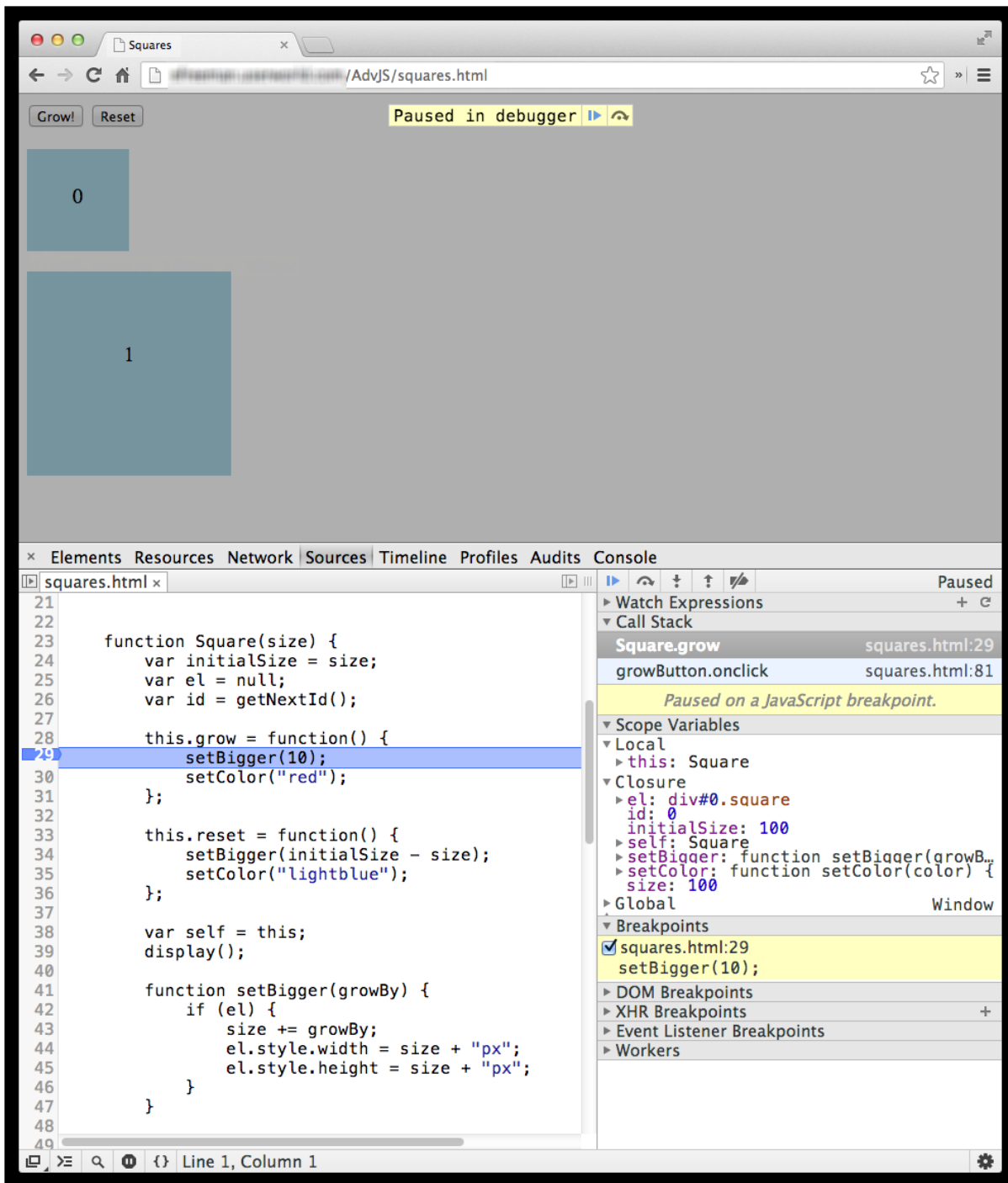
```
        return squares.length;
    }
    return 0;
}
}

window.onload = function() {
    var square1 = new Square(100);
    var square2 = new Square(200);

    var growButton = document.getElementById("growButton");
    growButton.onclick = function() {
        square1.grow();
        square2.grow();
    };
};
</script>
</head>
<body>
<form>
    <input type="button" id="growButton" value="Grow!">
</form>
<div id="squares"></div>
</body>
</html>
```

The closure created by the **Square()** constructor is a little less obvious, but it's there. In **this.grow()**, we refer to two nested functions, **setBigger()** and **setColor()**. Both are nested functions which means they are *local* variables in the **Square()** constructor. Just like any other kind of local variable, like **initialSize** or **id**, the values of these functions will disappear once **Square()** completes executing.

Because we reference these functions in **this.grow()**, the functions are added to the closure for the **this.grow()** method. In addition, any local variables that are in scope for these two functions are also added to the closure. Why? Because those values might be needed when we call **square1.grow()** and **square2.grow()**, otherwise, we'd get a reference error. So both of the function values that are used directly by **this.grow()**, as well as any other variables in scope for those two functions, are added to the closure. You can inspect the closure by adding a breakpoint to one of the lines of code in **this.grow()**. When you click the **Grow** button to call the **this.grow()** method of the square, you'll hit the breakpoint, and you'll be able to see the closure:



Whenever you create a function that references variables from the surrounding context, a closure is created. If you return that function from a function, or assign it to an object property, so the function is available outside of the context within which it was created, the closure comes along with the function. This means the function can "remember" the values of the variables it references. This is where the closure gets its name: a closure "closes" over the variables in scope when the function is created so it can keep them available for the function later, after the original context disappears. Think of closures as functions plus scope. If you understand scope, you'll understand closures too.

Note that closures aren't necessary for global variables, because global variables have global scope. They are available everywhere in your code, so there's no need to "remember" them in a closure.

The primary use for closures is to create private data, like we did with the counter example and with the squares example. You'll see closures used this way frequently (for example, in libraries like jQuery and Backbone.js).

Closures are notoriously tricky to wrap your head around, so take some extra time to review the lesson again and make sure you've got it. Use the Chrome console to inspect the closures you create to help you understand what's going on.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# The Module Pattern

## Lesson Objectives

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

- use an Immediately Invoked Function Expression, IIFE, to create a local scope for your code.
- explain how variables are accessible after the IIFE has completed by using a closure.
- use the Module Pattern in the implementation of jQuery.
- use the Module Pattern to create modules of code.

## Module Pattern

The Module Pattern has emerged as one of the most common patterns you'll see in JavaScript. It's used by most JavaScript libraries and plug-in scripts, including libraries like Backbone.js, jQuery, YUI, Prototype, and more. You can use the pattern to keep code organized, reduce the number of globals you use, encapsulate structure and behavior, and provide a simple API for your objects. All this is possible because of closures, which you learned about in the previous lesson. In this lesson, we take a look at the Module Pattern: what it is, how it works, how it's related to closures, and how it can help you organize your code.

## IIFE or Immediately Invoked Function Expressions

To understand the Module Pattern, you need to know how closures work (you've done that), how to use objects with closures to create private and public data (you've done that), and what it means to create a public API for an object (you've done that too). The only piece you don't know yet (although we did see one in the previous lesson) is the Immediately Invoked Function Expression.

Let's take a look at the simplest kind of IIFE you can create:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> IIFE </title>
  <meta charset="utf-8">
  <script>
    (function() {
      var x = 3;
    }) ();
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **iife.html**, and **Preview**. You won't see anything yet because so far, our IIFE doesn't do much. We need to make it do something, but first, let's explore exactly what's going on in the code:

### OBSERVE:

```
(function() {
  var x = 3;
}) ();
```

Let's break down this code into three parts. First, look at the **function expression**. This is an anonymous function (a function without a name), and all it does right now is initialize a variable **x** to 3.

Second, we have **parentheses** around the function. We put them there because we want to **execute the function, immediately**, which is the third part. We need the **parentheses** around the function to make it a

*function expression* rather than a *function declaration*, because JavaScript will create a function declaration automatically when it sees the **function** keyword when you use it at the global level. If you try to execute a function declaration immediately, you'll get a syntax error (try it and see: remove the parentheses and you'll get a syntax error). By putting the **parentheses** around the function, we create a function expression, which we can then execute immediately by adding **parentheses** after the function.

That code is *almost* the same as this code:

```
OBSERVE:
function foo() {
    var x = 3;
}
foo();
```

Here, we have a **function declaration** for the function **foo**. Then we **call the function** immediately after defining it.

Both these pieces of code accomplish essentially the same thing: they create a context (or scope) for the variable **x**. **x** is a local variable; it is available only inside the function scope. However, there is one key difference: in the first version, we never *name* the function, so no changes are made to the global object. The code executes without affecting the global object at all (it doesn't add any new variable or function definitions). In the second version, we do change the global object: we add the name **foo** to it and the value of **foo** is set to a function.

The first version of the code is an IIFE. The IIFE is used to create a context with a function within which you can declare variables, define functions, and execute code without affecting the global variables in your JavaScript. An IIFE is named as such because we call (invoke) the function in the same expression where we define it. In other words, we call the function expression immediately.

An IIFE can be pretty useful. For instance, you could create an IIFE that sets up a click handler for an element in your page. Modify **iife.html** as shown:

```
CODE TO TYPE:
<!doctype html>
<html>
<head>
  <title> IIFE </title>
  <meta charset="utf-8">
  <script>
    (function() {
      var x = 3;
      var message = "I've been clicked!";
      window.onload = function() {
        var div = document.querySelector("div");
        div.onclick = function() {
          alert(message);
        };
      };
    }) ();
  </script>
</head>
<body>
  <div>click me!</div>
</body>
</html>
```



and **Preview**

The words **click me!** appear in your page. Click on the text, and you see the "I've been clicked!" alert.

We've accomplished some work in the page (set up a click handler), but again, we've done it without adding anything new to the global object except for a value for the **window.onload** property. However, the other variables, **x**, **message**, and **div** are all private to the IIFE, and disappear once the function has finished executing when the page is loaded.

We can, however, click on the text and see the message *after* the IIFE has long gone. How? With a closure, of



course! Whenever we create a function that references variables defined in the surrounding context of that function, and make that function available for use outside of that context, a closure that contains the values of the variables that function needs comes with it so the function "remembers" those values.

Inside our IIFE, we assign a **click handler function** to the **onclick** property of the `<div>` element. This function references the variable **message**, and because the function is available *after* our IIFE goes away (because we saved it in the `<div>` element's `onclick` property, and the `<div>` element doesn't go away), we get a closure along with the function. That closure contains the variable **message**.

OBSERVE:

```
(function() {  
    var message = "You've been clicked!";  
    window.onload = function() {  
        var div = document.querySelector("div");  
        div.onclick = function() {  
            alert(message);  
        };  
    };  
})();
```

IIFEs are useful for getting work done while having minimal effect on the global scope. Additionally, you can use them to set up code to run later by assigning values to properties of objects, like the `<div>` element, that do stick around after the IIFE is gone.

## The Module Pattern

Now you know everything you need to know in order to use the Module Pattern. Let's look at an example of a small program that is structured using this pattern. Create a new file as shown:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Module Pattern </title>
  <meta charset="utf-8">
  <script>

    var counterModule = (function() {
      var counter = 0;

      return {
        increment: function() {
          counter++;
        },
        decrement: function() {
          counter--;
        },
        reset: function() {
          counter = 0;
        },
        getValue: function() {
          return counter;
        }
      };
    }) ();

    window.onload = function() {
      counterModule.increment ();
      counterModule.increment ();
      counterModule.decrement ();
      counterModule.increment ();
      console.log(counterModule.getValue ());
      counterModule.reset ();
    };

  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **module.html**, and **Preview** . The value **2** displays in the console.

Let's examine this code more closely. First, we use an IIFE to create an object that manages a variable, **counter**. Unlike our previous IIFE example, we actually return a value from the function: an object that contains four methods to manage the counter. However, the counter is not part of the object that's returned. Instead, it's a local variable to the IIFE, which means that it's a private variable. Because the object's methods reference this variable, when the object is returned, we get a closure with each of the methods that contains the **counter** variable (and note that each closure references the *same* variable, so if one of the methods changes the value of **counter**, it will change for the entire object).

Next, we name the object that's returned. We give it the name **counterModule**. This is a global variable so it's available to the rest of our program to use, but note that it's the only global variable we create (other than setting the **window.onload** property to a function). The counter itself is private and accessible only using methods in the **counterModule** object.

This is the point of the Module Pattern: to minimize the number of global variables and functions you create in your program (preferably limiting the number to just one global object that contains everything else you need). The object you return from your IIFE contains public data and public methods, and all the private data is in the closure associated with the object's methods. So the object acts as an API to all the functionality the object provides to your page. This is the "module": an independent set of functionality that contains everything necessary to execute one aspect of the overall desired functionality in the page. In this example, our "module" is the counter: everything you need for the counter is encapsulated within the **counterModule** module.

As you can see, the concepts used in the Module Pattern are all concepts we've seen before: public and

private data, an API, and closures. The pattern a way to describe how to use these concepts together to structure your code a certain way. A pattern is a design; it's not an implementation. It's a guideline for how to structure your code to achieve a goal. In this case, that goal is to create a context for a set of functionality that is accessible to the page globally, but has minimal impact on the global variables in the page.

## Using the Module Pattern with JavaScript Libraries

The Module Pattern is particularly useful for JavaScript libraries and widgets, because it allows you to combine code from several different sources, knowing that it's unlikely you'll overwrite variables from another library by mistake. Because the module object provides a public API for managing private data, it's also unlikely that you'll do something disastrous by accessing a variable in a way you shouldn't (if you use the API correctly).

As an example, look at how [jQuery](#) is structured. The actual source code has some additional complexities we won't go into here, but if you look at the snippet of source code below, you'll see that the authors of jQuery use the Module Pattern to structure the code for the library. The internals of jQuery are implemented as private variables and methods, and the library functionality for you to use is exposed through the public methods in the jQuery object (also named \$). In this case, once the jQuery object has been set up with everything it needs inside the IIFE, rather than returning that object, the jQuery object is assigned to two properties in the global object: **window.\$** and **window.jQuery** (so you can refer to the jQuery module using either name):

```
OBSERVE:
(function( window, undefined ) {

    // lots of code here

    // Define a local copy of jQuery
    jQuery = function( selector, context ) {
        // The jQuery object is actually just the init constructor 'enhanced'
        return new jQuery.fn.init( selector, context, rootjQuery );
    },

    // lots more code here...

    // Here is where we add the jQuery object as a global variable
    // so you can access all the public properties and methods
    window.jQuery = window.$ = jQuery;

})( window );
```

Don't worry about the details of the code above (which is just a tiny snippet taken from the current version of jQuery). Just notice that the structure of the library uses the Module Pattern. Almost every JavaScript library out there uses this pattern.

If you're using multiple libraries, you might find it useful to structure your own code using the module pattern, and *import* the libraries into your module like this:

```
OBSERVE:
var myModule = (function(J$, U$) {

})(jQuery, _);
```

Here, we pass two arguments into our IIFE: the **jQuery** object (which we get when we link to the jQuery library) and the **\_** object, which is the name of the global object for the [Underscore](#) library. We can *alias* these two library objects by using different names for the parameters in the IIFE: **J\$** for jQuery and **U\$** for Underscore. You might just like these names better; but sometimes you can do this to avoid name clashes within your module.

## A Shopping Basket Using the Module Pattern

Here's another example of using the Module Pattern—a shopping basket module:

**CODE TO TYPE:**

```
<!doctype html>
<html>
<head>
  <title> Shopping Basket: Module Pattern </title>
  <meta charset="utf-8">
  <script>
    var basket = (function() {
      var basket = {};
      var items = [];

      //
      // Add a new item to the basket.
      // If item already exists, increase the count of existing item.
      // Returns: number of that item in the basket.
      //
      function addItem(item, cost) {
        for (var i = 0; i < items.length; i++) {
          if (items[i].name == item) {
            items[i].count++;
            return items[i].count;
          }
        }
        items.push({ name: item, price: cost, count: 1 });
        return 1;
      }

      //
      // Remove an item from the basket
      // If item has more than 1 in basket, reduce count.
      // If no more items left after removing one, remove item completely.
      // Returns: number of that item left or -1 if item you tried
      // to remove doesn't exist.
      //
      function removeItem(item) {
        for (var i = 0; i < items.length; i++) {
          if (items[i].name == item) {
            items[i].count--;
            if (items[i].count == 0) {
              items.splice(i, 1);
              return 0;
            }
            return items[i].count;
          }
        }
        return -1;
      }

      //
      // Compute the total cost of items in the basket.
      //
      function cost() {
        var total = 0;
        for (var i = 0; i < items.length; i++) {
          total += items[i].price * items[i].count;
        }
        return total;
      }

      basket.addItem = function(item, cost) {
        var count = addItem(item, cost);
        console.log("You have " + count + " of " + item + " in your basket."
);
      };

      basket.removeItem = function(item) {
        var count = removeItem(item);
        if (count >= 0) {

```

```

        console.log("You have " + count + " of " + item + " left in your
basket.");
    } else {
        console.log("Sorry, couldn't find " + item + " in your basket to
remove.");
    }
};
basket.cost = function() {
    var totalCost = cost();
    console.log("Your total cost is: " + totalCost);
};
basket.show = function() {
    console.log("==== Shopping Basket =====");
    for (var i = 0; i < items.length; i++) {
        console.log(items[i].count + " " + items[i].name + ", " + (items
[i].price * items[i].count));
    }
    console.log(" ");
};

return basket;
})();

window.onload = function() {
    basket.addItem("broccoli", 1.49);
    basket.addItem("pear", 0.89);
    basket.addItem("kale", 2.38);
    basket.addItem("broccoli", 1.49);
    basket.show();

    basket.cost();

    basket.removeItem("broccoli");

    basket.cost();
};

</script>
</head>
<body>
</body>
</html>

```



Save this in your **/AdvJS** folder as **basket.html**, and **Preview** . In the console, this output is displayed:

#### OBSERVE:

```

You have 1 of broccoli in your basket.
You have 1 of pear in your basket.
You have 1 of kale in your basket.
You have 2 of broccoli in your basket.
==== Shopping Basket =====
2 broccoli, 2.98
1 pear, 0.89
1 kale, 2.38

Your total cost is: 6.25
You have 1 of broccoli left in your basket.
Your total cost is: 4.76

```

In our module, we have a couple of private variables: **basket** (the object we return as the value for the module), and **items** (an array that holds the items in your basket). We also have a few private functions that handle the functionality of managing the shopping basket: **addItem()**, **removeItem()**, and **cost()**. Once the module is created, none of these private variables or functions will be available to the user of the module, except through the public API which is created by returning the **basket** object from the IIFE and storing the

resulting value in the **basket** global variable.

Notice that we use the same name for the local variable for the **basket** object and the global variable to store the finished module. This is perfectly fine.

The public API is the collection of methods in the **basket** object: **basket.addItem()**, **basket.removeItem()**, **basket.cost()**, and **basket.show()**. These methods can be used by code that uses the **basket** module.

Look through the code and make sure you understand how it works and how it's structured using the Module Pattern. Keep in mind that the pattern is a *design guideline*, not a specific implementation, so it's expressed in code in different ways depending on the needs of the specific module you're implementing.

## Why Not Just Use an Object Constructor?

So, why would you use the Module Pattern when you could just use an object constructor and achieve the same thing, like this?:


```
CODE TO TYPE:
<!doctype html>
<html>
<head>
  <title> Counter Constructor instead of Module Pattern </title>
  <meta charset="utf-8">
  <script>
    function CounterModule() {
      var counter = 0;

      this.increment = function() {
        counter++;
      };
      this.decrement = function() {
        counter--;
      };
      this.reset = function() {
        counter = 0;
      };
      this.getValue = function() {
        return counter;
      };
    };

    var counterModule;
    window.onload = function() {
      counterModule = new CounterModule();
      counterModule.increment();
      counterModule.increment();
      counterModule.decrement();
      counterModule.increment();
      console.log(counterModule.getValue());
      counterModule.reset();
    };

  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **module2.html**, and **Preview** . In the console, the same output displays as with our previous version of the counter module: 2.

Here, we use a constructor function, **CounterModule()**, to create exactly the same kind of object that we did before: an object with some public methods to manage the counter, which is a private variable (and accessible only through those public methods).

The difference between the two approaches is in the way we create the counter. If we use a constructor function, we need to create a `counterModule` using `new CounterModule()` (which is fine). Using `new` to create the object or using the Module Pattern to create the object accomplishes essentially the same thing.

You'd use the Module Pattern when you want *just one* version of the object. When we use the Module Pattern, we know that the user of the module can't instantiate multiple instances of the object, because there's no constructor. When we use a constructor, we can get many instances of the object. In the case of libraries like jQuery, Underscore, and others that use the Module Pattern, we know we'll only want one of each of these objects (having more would be pointless). So, whether you use the Module Pattern or an object constructor really depends on how you plan to use the object (or objects) you create. Do you need just one? Use the Module Pattern. Do you need many? Use a constructor.

In this lesson, you learned about the Module Pattern: a design guideline for how to structure your code to reduce the number of global variables you use, manage private data and functionality, and create a public API (through one global object) to access that private data and functionality. Because JavaScript has one global object that's shared by all the code that you write in a page, as well as any code you link to (both external libraries and your own additional code), this is a popular pattern that helps to reduce name clashes and keep the global namespace "clean." This is particularly important for large projects where you may not always know the names that are being used in other parts of the code.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# The JavaScript Environment

---

## Lesson Objectives

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

- explore your browser's JavaScript extensions.
  - distinguish between the core language and the extensions provided by the environment in which your code runs.
  - compare how the browser runs your code depending on where you put that code in a page.
  - describe and distinguish the two phases the browser uses to execute JavaScript code.
  - explain how the event loop works.
  - create and handle multiple events.
- 

## JavaScript Runs in an Environment

In this course, we've been focusing on the language features of JavaScript, but there's more to it because JavaScript runs in an *environment*. Most of the time, that environment is the browser. We've seen a little of the interaction between JavaScript and the browser when we used JavaScript to update the DOM by adding new elements or styling elements, or getting user input from a form. In this lesson, we look closer at how JavaScript interacts with its environment, and things you need to be aware of when running JavaScript in the browser.

### The Core Language, and the Environment's Extensions

JavaScript was created in 1995 (eons ago in internet time) specifically to run in the Netscape browser; JavaScript is still primarily a language for the browser. It has gotten beyond the browser though, most notably as a scripting language for PDF documents, OpenOffice, DX Studio, Logic Pro to name a few. In addition, JavaScript can now be used as a server-side language running in environments like Node.js.

JavaScript runs in an environment, and that environment is usually the browser. Depending on the environment in which JavaScript is running, there will be ways you can alter the language to manipulate that environment. If you're running JavaScript in the browser, you get extra "stuff" along with the language basics, that allows you to get data from the page, manipulate the content of the page, and even change the style of the page.

You can think of JavaScript as a language that's composed of two parts: the *core* language and the *extensions* that are supplied by the environment in which it's running. The *core* of the language are the syntax and semantics that control actions like how you define variables and functions, how you write loops, how you call functions, how scope works, and so on.

The *extensions* to the language are the parts that are supplied by the environment. Typically these parts are objects that provide a JavaScript interface for you to use in that environment. For the browser, this refers to elements like the **document** object, the **window** object, and all the properties and methods that come along with those objects.

Inspect these objects in the console:

#### INTERACTIVE SESSION:

```
> window
Window {top: Window, window: Window, location: Location, external: Object, chrome: Object}
```

Try this in Chrome. Twirl down the arrow next to the result, and you'll see a long list of the various properties of the **window** object:





Every browser will give you some representation of the **window** object if you type this into the console, but they might be a little different from one another.

Now try the **document** object:

```
INTERACTIVE SESSION:
> document
#document
  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
  <html>...</html>
```

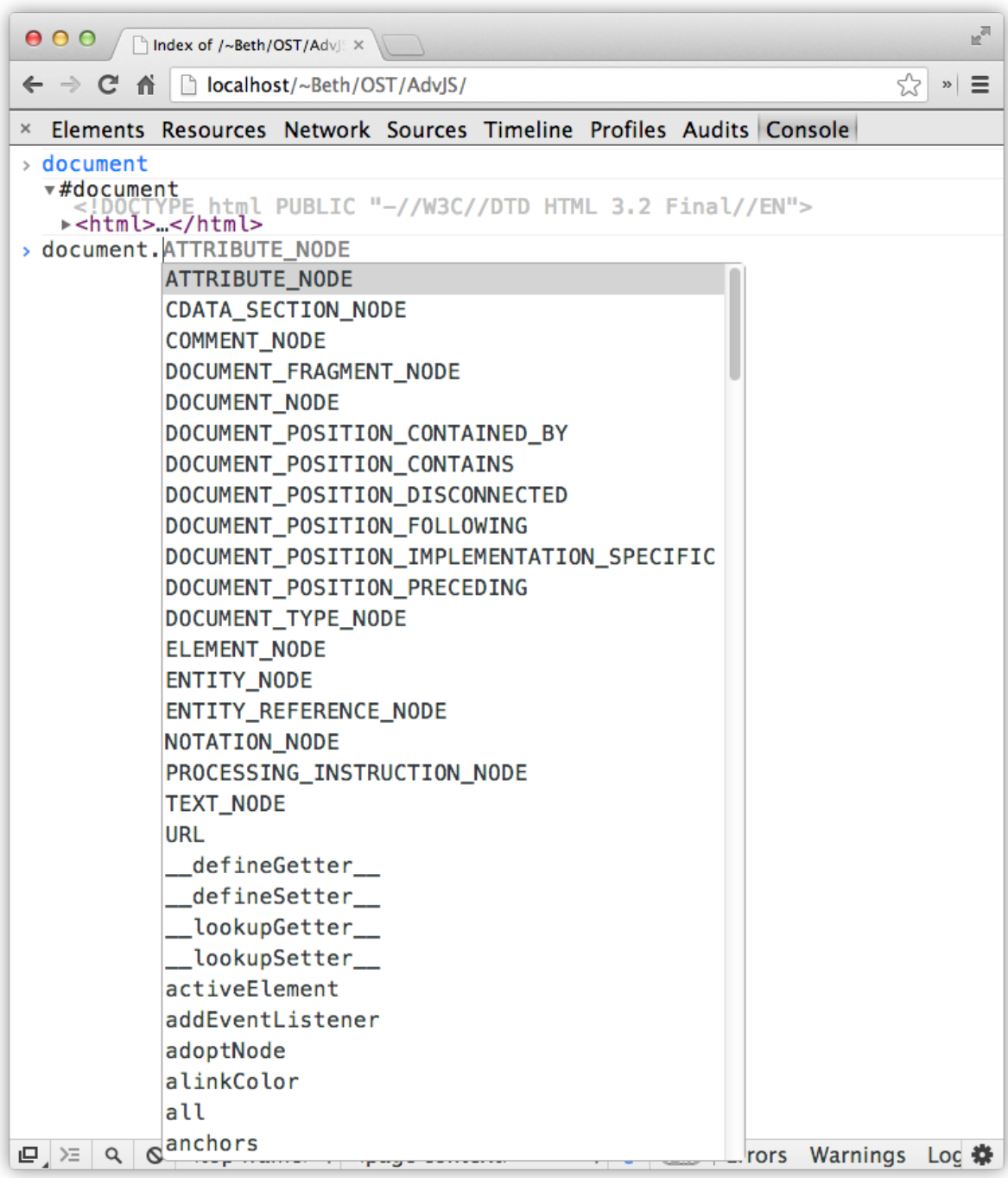
Again, twirl down the arrow next to **#document** so you can see a display of the **document** object. The **document** object represents your web page, so when you open it up, you see the content of your page.

**document** has lots of properties and methods you can use to access the content of your page, like

`getElementById()` and others. To see which properties `document` has, type "document" with a period following it:

```
INTERACTIVE SESSION:
> document.
```

If you do this in Chrome, Safari, Firefox or IE, you'll see a pop up window that shows you all the properties of `document` that you can type next:



Scroll down through the list and you'll see many properties you're familiar with, as well as many you're not familiar with yet.

The `document` object is a property of the `window` object:

#### INTERACTIVE SESSION:

```
> window.document
#document
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<html>...</html>
```

All of the browser-related things you'll do with JavaScript are properties of the **window** object. We say the **window** object is the *global* object or *head* object, because it gives you access to all the other objects you'll use to manipulate the browser environment. Try typing the names of these properties in the browser and see what you get (we're showing results from Chrome here):

#### INTERACTIVE SESSION:

```
> window.localStorage
Storage {2269ae85-adeb-40cb-aeef0-c43e9b4940ea_popup_openPosition: "{"top":184,
"left":270}", 235486a4-943d-45a7-a1f4-31d1b4d6bae1_popup_openPosition: "{"top":1
84,"left":314}", 3437639f-10f2-4bd7-8e24-2314e5beeb6d_popup_openPosition: "{"top
":184,"left":270}", 3af686d4-ef0e-4ba4-a302-81bf2a8c23d6_popup_openPosition: "{"
top":33,"left":2}", 46c2259c-fa5e-4c41-8bf7-3973fc2f678e_popup_openPosition: "{"
top":184,"left":270}"...}
> window.navigator.geolocation
Geolocation {getCurrentPosition: function, watchPosition: function, clearWatch
: function}
> window.JSON
JSON {}
```

Remember that because the **window** object is the global object, it's also the *default* object, so to access a property of the **window** object you don't have to type **window**. For instance, you can write **window.alert()** or just **alert()**, **window.JSON**, or just **JSON**.

You can also create and inspect elements right in the console to see which properties are supported by the various element objects in the browser. For instance, if you want to see the properties supported by the `<video>` element, try this:

#### INTERACTIVE SESSION:

```
> var media = document.createElement("video")
undefined
> media.
```

We created a `<video>` element using the **document.createElement()** method. Once we have that element (in a variable, it's not added to the page), we can inspect it by typing "media" followed by a period. Just like before, a popup window appears with all the various properties that the `<video>` element supports. There are some different properties and methods in the list that aren't supported by other elements, or by the **document** object. Because a **video** object is a DOM object, it will inherit many methods that all element objects have, but it has a few, like **autoplay** and **loop**, that are unique to video (and audio, which has many of the same properties as video).

## How the Browser Runs JavaScript Code

When JavaScript was first added to browsers, it was purely an *interpreted* language. To run the JavaScript in your web page, the browser begins interpreting and executing your JavaScript, from the top down, as the page is loaded. Each line of your code is parsed by other code internal to the browser and then evaluated. The browser's runtime environment contains all of your variables, functions, and so on, so if you declare a variable `x` and give it the value 3, the browser creates a bit of storage in that runtime environment, and stores the value 3 there.

Interpreted languages are typically slower than compiled languages, like C, C++, Java, and C#, because they are not converted to machine code or optimized before they are run. The browser interprets each and every line of code as it gets to the next line, and has to parse each line of code just as you've written it.

For this reason, JavaScript in the browser has been notoriously slow. However, as developers began to expand the way web pages are used, and create web pages that are more like applications than static documents, browser developers began to see a huge advantage to making JavaScript faster. Think of Google Maps: we want maps to run fast so we can scroll around in the map, zoom in and out, and have the page respond quickly.

Browser developers began to build JavaScript engines into browsers that compile JavaScript code (using "Just In Time," or JIT, compilers), and turn it into a special code that could be run faster by the browser. For example, Chrome's V8 JavaScript engine compiles your JavaScript into machine code before the browser executes the code. As the browser compiles your code, it can make optimizations so that the code will run even faster.

Browsers still run your code top down (as most runtime environments do), so how you write your code hasn't changed. The way the browsers deal with your code has changed though, which has resulted in huge speed increases when you run JavaScript, so now you can write huge applications in your web pages that run fast (think of applications like Google maps and mail, Facebook, Netflix and Hulu, Evernote, Vimeo and YouTube, and many more).

**Note** To learn more about how JavaScript engines in browsers work, check out the links on the [JavaScript Engine](#) Wikipedia page.

## Including JavaScript in Your Page

There are two ways to include JavaScript in your page: as embedded script in an HTML page (using the `<script>` element), and as a link to an external script. If you include your script (as an embedded script or as a link) at the top of your page, typically in the `<head>` element, the JavaScript will be executed *before* the browser interprets your HTML:

OBSERVE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    var x = 3;
  </script>
</head>
<body></body>
</html>
```

or

OBSERVE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script src="external.js"> </script>
</head>
<body></body>
</html>
```

Place your code at the end of your page, like this:

#### OBSERVE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
</head>
<body>
<div>
  Other HTML here
</div>
<script>
  var x = 3;
</script>
</body>
</html>
```

The code will run *after* the rest of your page has loaded and the browser has interpreted the HTML.

We used to recommend that you add your JavaScript in the <head> of your document, but recently more developers are recommending that you add your JavaScript at the bottom, just before the closing </body> tag. As JavaScript gets larger (for more complex pages), it takes longer to download, parse, and execute the JavaScript, so the user has to wait longer to see the web page. Either way will work, but if you're writing a complex page with large JavaScript files, you may want to test your page to see if including the JavaScript at the bottom of the file leads to a better user experience.

An advantage of having your JavaScript in an external file is that if the JavaScript is used by multiple web pages, the browser will *cache* the JavaScript file, so it doesn't have to be downloaded multiple times. This is particularly important for library files, like jQuery or Underscore.js, which are typically used for a whole web site rather than just one page. If you use a well-known URL for the library (like the site's hosting URL, or even a [URL on Google's servers](#)), it's likely that the user's browser already has that file cached, which will reduce the download time even more.

To see the difference between including your code in the <head> of your page and at the bottom, try this:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Script Testing </title>
  <meta charset="utf-8">
  <script>
    var div = document.getElementById("div1");
    div.innerHTML = "Testing when the browser loads a script";
  </script>
</head>
<body>
  <div id="div1"></div>
</body>
</html>
```




Save this in your /AdvJS folder as **testLoad.html**, and . Open the console. You see a message, "Uncaught TypeError: Cannot set property 'innerHTML' of null."

Now move the script to the bottom of the page:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Script Testing </title>
  <meta charset="utf-8">
  <script>
    var div = document.getElementById("div1");
    div.innerHTML = "Testing when the browser loads a script";
</script>
</head>
<body>
  <div id="div1"></div>
  <script>
    var div = document.getElementById("div1");
    div.innerHTML = "Testing when the browser loads a script";
  </script>
</body>
</html>
```



and . Now there's no error in the console, and the page displays the message.

Because the browser evaluates the page from the top down, in the first version, the DOM is not ready when the browser executes your script. In the second version, the DOM has been built and the `<div>` that you are modifying exists in the DOM, so the script works. If you prefer to put your script in the `<head>` of your document, make sure any code that uses or manipulates the DOM is called from within the **`window.onload`** event handler. If you've been working with JavaScript for a while, you already know this, but it's worth reiterating because it's important.

## The JavaScript Event Loop

Once the browser has executed your code from the top down as it loads your page, it enters into an *event loop*. This loop is internal to the browser. It is basically a loop that waits for events to occur. If you move your mouse, or click on an element, or request data from another website, an event will be generated. There are internal browser events as well. For instance, when the browser has finished loading the page, it generates the "load" event, which will cause your **`window.onload`** event handler to execute, if you've defined one.

When an event happens, the browser checks to see if you've defined an event handler for that event; if you have, that function is run. Once the function is complete, the browser begins the event loop again. This event loop continues as long as the web page is loaded.

In this phase of execution (that is, the phase when the browser is waiting for events and executing event handlers), events can happen at any time, so your event handler functions run when they're needed, not at any particular time or in a particular sequence.



You can disrupt the event handling process by writing code that takes up too much of the processing power of your browser. Take a look at the code below.

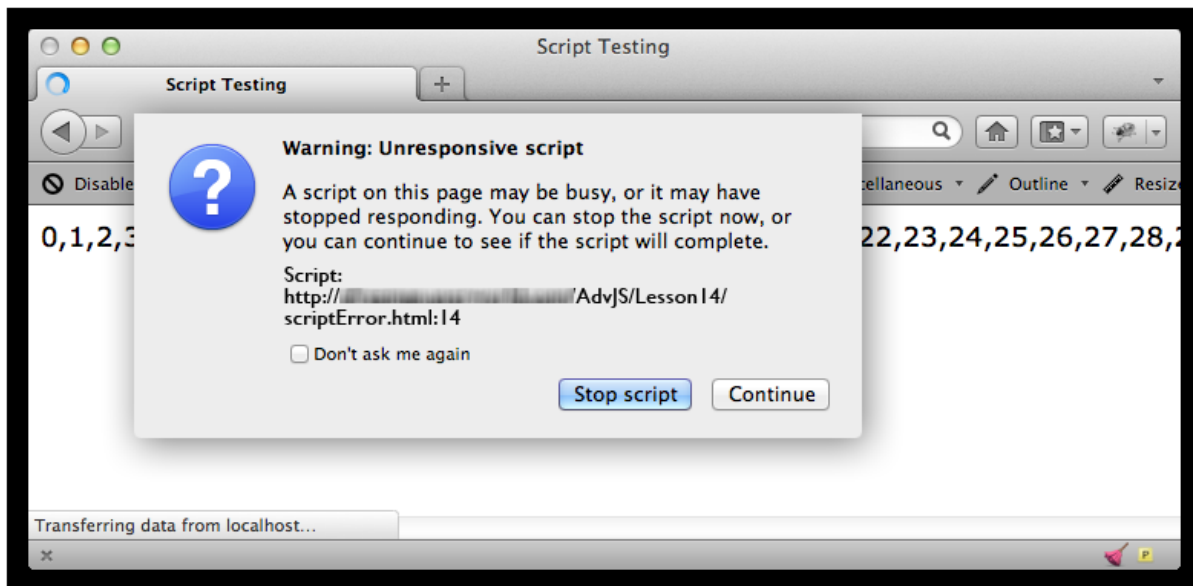
### **WARNING**

Feel free to type in the code below and try it, but be warned—you will probably need to close the browser entirely to stop the code. If you do want to try it, we recommend using a different browser from the one you're using to read the lesson.

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Script Testing </title>
  <meta charset="utf-8">
  <script>
    window.onload = function() {
      document.onclick = function() {
        alert("You clicked on the web page!");
      }
      var div = document.getElementById("div1");
      for (var i = 0; i < 100000000000; i++) {
        div.innerHTML += i + ",";
      }
    }
  </script>
</head>
<body>
  <div id="div1"></div>
</body>
</html>
```

 Save this in your **/AdvJS** folder as **scriptError.html**, and **Preview** . You may get an "unresponsive script" dialog box, like the one below (from Firefox):



If you do, choose the option to stop the script and you may get control of your browser back. However, your browser may become entirely unresponsive. In that case, you can force quit the browser on the Mac by pressing **Option+Command+Escape** (all at once), selecting the browser that's unresponsive, and choosing **Force quit**. In Windows, right-click in the taskbar, select **Start Task Manager**, select the browser application, and select **End Task**.

Let's step through the code to see what's happening:

## OBSERVE:

```
window.onload = function() {
  document.onclick = function() {
    alert("You clicked on the web page!");
  }
  var div = document.getElementById("div1");
  for (var i = 0; i < 100000000000; i++) {
    div.innerHTML += i + ",";
  }
}
```

First, we're running all the code in the **window.onload** event handler. This runs once the browser has completed loading the page, so we can access the DOM safely.

We set up a **click event handler on the document**. That means anywhere you click on the page will trigger this event. Once this is set up, we **get the "div1" object from the page**, and then begin a **loop** that will add successive integers to the content of the <div> using **innerHTML**. However, the loop end point is a very large number. Even for fast computers, this loop is going to take a long time.

Now, try clicking on the page if you want. The browser is so busy executing the loop that it is unlikely to recognize the event for a while (if ever) before you see a browser error.

Change the very large number above to 10,000:

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Script Testing </title>
  <meta charset="utf-8">
  <script>
    window.onload = function() {
      document.onclick = function() {
        alert("You clicked on the web page!");
      }
      var div = document.getElementById("div1");
      for (var i = 0; i < 10000; i++) {
        div.innerHTML += i + ",";
      }
    }
  </script>
</head>
<body>
  <div id="div1"></div>
</body>
</html>
```



and **Preview**

Click on the web page as soon as it loads. You might have to wait a little while, but you'll eventually get an alert.

## The Event Queue

So, it still takes a little while for the browser to get through the loop, but eventually it does, and it responds to your click. Even if you click on the page *before* the loop is complete, the browser doesn't forget the event, because whenever an event occurs in the browser, that event is added to an *event queue*. The browser handles events in order as they occur. The event queue ensures that even if multiple events are happening, and some of the handlers for those events take a while to execute, the browser eventually gets to all of them (unless an event handler takes so long that it causes the browser to display an unresponsive script dialog, or if an event handler causes an error so that your browser stops executing your script altogether).

## Asynchronous Programming

Suppose you write a script that sets up a click handler for a mouse click, and also sends a request to get more data using XHR (XMLHttpRequest, also known as Ajax). Your request to get more data might be




initiated right away, but if it takes a while to get the data, the event handler that will be called once the data is received won't execute for a while. In the meantime, you can click your mouse a bunch of times and your mouse click event handler will execute.

The XHR request that you created to get more data is executed *asynchronously*. That is, the browser allows you to do other things while it's waiting to get the data from the XHR request. In fact, while it's waiting for the data, the browser goes back to executing the event loop so it can respond to other events. When the data you requested with XHR is finally retrieved, your XHR event handler will be executed. Compare the way the browser handles a synchronous loop, like we created above, with the way it handles an asynchronous request like XHR: The loop jams up the event loop, while the XHR request does not.

Here's a quick example that uses XHR to fetch data for your application. It also sets up a click handler:


```
CODE TO TYPE:

<!doctype html>
<html>
<head>
  <title> XHR Asynchronous Request </title>
  <meta charset="utf-8">
  <script>
    window.onload = function() {
      document.onclick = function() {
        alert("You clicked on the web page!");
      }
      var request = new XMLHttpRequest();
      request.open("GET", "data.json");
      request.onreadystatechange = function(response) {
        var div = document.getElementById("div1");
        if (this.readyState == this.DONE && this.status == 200) {
          if (response != null) {
            div.innerHTML = this.responseText;
          }
          else {
            div.innerHTML += "<br>Error: Problem getting data";
          }
        }
        else {
          div.innerHTML += "<br>Error: " + this.status;
        }
      }
      request.send();
    }
  </script>
</head>
<body>
  <div id="div1"></div>
</body>
</html>
```

If you want to try this code and see these events in action,  save the file in your **/AdvJS** folder as **eventQueue.html**. Create a file named **data.json** in the same folder. You can put any JSON data in the file, like this:

```
OBSERVE:

[ "test data" ]
```

**Preview**  Preview the **eventQueue.html** file. [ "test data" ] appears in the web page. You can click on the page to see the alert.

Our data is small so it doesn't take much time to retrieve, so the browser will load your JSON data far too fast for you to be able to click on the page before it does. Still, you can see from the code that we're setting up two events: one will occur whenever you click the mouse, and the other will occur when the data in the file "data.json" has been retrieved. We have no way of knowing which will occur first (but we can make an educated guess because you can't click the mouse fast enough). If you never click the mouse, then the

mouse click event will never happen.

If you replace the small amount of data here with a much larger amount of data, you might be able to delay the loading long enough to click the mouse. Give it a try if you want.

Remember, JavaScript in the browser is *event-driven*; once the browser has loaded and executed your code in the first phase (as it loads the page), the second phase responds to events, which makes your web pages interactive.

Also keep in mind that some actions, like XHR requests, are executed *asynchronously*. We set up XHR event handlers like we do other kinds of event handlers: by assigning a function to a property on an object, in this case, the XMLHttpRequest object. The event handler is called when the data requested by the XHR request has been returned to the browser. We don't know precisely when the event will occur, and because the browser doesn't stop the event loop while it's waiting for the data, we can continue to interact with the browser and run other code.

## JavaScript in Environments Other Than the Browser

You're using JavaScript in the browser, but much of what you've learned in this course applies to JavaScript in other environments as well..

For instance, if you're writing a JavaScript script for Photoshop, the main entry point for access to the Photoshop environment is the **app** object. For more about writing JavaScript for Photoshop, check out the links here: <http://www.adobe.com/devnet/photoshop/scripting.html>. Of course, other Adobe products also offer scripting capabilities.

Another JavaScript environment that is rising in popularity is Node.js. Node.js allows you to run JavaScript at the command line, and you can use Node.js to serve web pages and run web services. If you download and install Node.js, you can experiment with it by running **node** at a command line. Interact with it just like you would the JavaScript console in the browser, except that you won't have all the built-in browser objects; instead, you'll have built-in Node.js objects. Here's a sample interactive session from my own computer (you won't be able to reproduce this unless you install Node.js, which is not necessary for this course; we're just showing this session in case you're interested):

### INTERACTIVE SESSION:

```
[elisabeth-robsons-mac-pro]% node
> var i = 3;
undefined
> i
3
> console.log("i is " + i)
i is 3
undefined
> require("os")
{endianness: [Function],
 hostname: [Function],
 loadavg: [Function],
 uptime: [Function],
 freemem: [Function],
 totalmem: [Function],
 cpus: [Function],
 type: [Function],
 release: [Function],
 networkInterfaces: [Function],
 arch: [Function],
 platform: [Function],
 tmpdir: [Function],
 tmpDir: [Function],
 getNetworkInterfaces: [Function: deprecated],
 EOL: '\n' }
> os.arch()
'x64'
> os.uptime()
28466
>
```

As you can see, the first part of this session looks just like a session in the browser's JavaScript console.

However, about half way through we write `require("os")`. This loads a Node.js *module*, which is just a library of JavaScript code (similar to if you linked to a library like jQuery from a web page). Once we've loaded this module, we have access to another object, `os`, that can give us information about the operating system on which we're we're running Node.js. JavaScript can't do this in the browser because the JavaScript in the browser executes in a *sandbox*: a special area of the browser that protects your system from any code that is downloaded and executed in the browser (which helps to prevent JavaScript viruses).

If you want to explore another JavaScript environment, you'll find that you can apply much of the information you've learned so far to that environment, but you'll also need to learn about the environment's extensions to understand how to use JavaScript within that environment.

The JavaScript objects that browsers provide for you to manipulate the browser are based largely on specifications written by the W3C (the World Wide Web Consortium). While most browser manufacturers have agreed that we are all better off if browsers support the same features (so we don't have to write different code for different browsers), various browser makers are always thinking up cool new features to add. Some of these features make it into the specifications and are implemented by the other browsers, and some do not.

All of the browser extensions we've talked about in this course are supported by the most recent versions of all the major browsers. To use cutting-edge features that are being added into browsers, test for those features, either by writing the test code yourself, or by using a JavaScript library like Modernizr. Testing for features rather than a specific browser ensures that your code will be forward-compatible: that is, if a user doesn't have a browser that supports that feature now, they may in the future, so your web applications will work for them.

Libraries, like jQuery and Underscore.js (and many others), can also help with browser compatibility. These libraries provide what are called "shims" that provide fallback behavior for features that aren't supported by all browsers.

To learn more about the specifications that (largely) determine which features browsers support, check out the W3C TR page (start at the DOM, DOM events, and JavaScript APIs TRs and go from there). In addition, each browser maker will have documentation on their sites about features supported by each of their individual browsers:

- [Safari Developer Center](#)
- [MDN Developer Network](#)
- [Chrome Developer Site](#)
- [IE Developer Site](#)
- [Opera Developer Site](#)

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# ECMAScript 5.1

## Lesson Objectives

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

- describe the most recent version of the standard on which JavaScript is based, ECMAScript 5.1.
- use strict mode to prevent common mistakes.
- explore new methods added in ES5.
- use object property descriptors to control access to objects.
- create objects based on your own prototypes.

## The ECMAScript Standard for JavaScript

JavaScript has a convoluted history, but the good news is that all the major browsers now (mostly) support a standard version of JavaScript. With a standard for the core language and standard for the browser extensions (as we talked about in the previous lesson), working with JavaScript is much better now than in the old days when we had to write different JavaScript for each different browser. What a mess that was!

The standard for the core JavaScript language is described in the ECMAScript specification, maintained by the ECMA International organization, an international, private (membership-based), non-profit standards organization for information and communication systems. The current version of the ECMAScript standard for JavaScript is 5.1. You can find the specification of the standard at <http://www.ecma-international.org/publications/standards/Ecma-262.htm> [PDF]. (An HTML version is also available).

Version 5.1 (which we often refer to as ES5) of the standard was completed in 2011; major browsers implement much of the standard today. Fortunately, version 5.1 is completely backward-compatible with the previous version (3.1; don't ask what happened to version 4), so you don't have to unlearn anything to learn the new stuff in version 5.1. Here's a handy compatibility table that describes browser support for various ES5 features. We'll experiment with some of these features in this lesson, so make sure you've got the most recent version of your favorite browser installed.

There are quite a few minor changes in the ES5 version of the language, a few interesting additions to Objects, and a new *strict* mode that we'll explore. Keep in mind that the language specification is for the *core* language, and does not refer to the browser extensions we explored in the previous lesson.


### Strict Mode

Despite what we just said about not having to unlearn anything for ES5, there are a few things you can do in JavaScript that you really shouldn't be allowed to do. (You haven't been doing those things in this course, but it's a good idea for you to know about them.) Some of these quirks have been *deprecated* in ES5. That means they are no longer supported in the language, but browsers will still let you do them (as part of the backwards compatibility with the previous version of the standard). Here's an example:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    myVar = "I'm not declared!";
  </script>
</head>
<body>
</body>
</html>
```



Save this in your **/AdvJS** folder as **strict.html**, and . Open the console and type the command shown:

## INTERACTIVE SESSION:

```
> myVar
"I'm not declared!"
```

You see the value of the string in the variable **myVar**. The problem with this code is that you didn't declare the variable **myVar**. As we discussed earlier in the course, this is not good. We always want you to declare your variables. Also, try to keep the number of global variables you use to a minimum.

ES5 has introduced a new mode, called *strict mode*, to help you catch these errors. When you are in strict mode, you can't assign a value to a variable that hasn't been declared. Here's how you put your code into strict mode:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    "use strict";
    myVar = "I'm not declared!";
  </script>
</head>
<body>
</body>
</html>
```



and



. In the console you see this error (or something similar if you're using a browser other than Chrome):

### OBSERVE:

```
Uncaught ReferenceError: myVar is not defined
```

In strict mode, you're not allowed to make this mistake. To fix the mistake, add the "var" keyword in front of the variable:

### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    "use strict";
    var myVar = "I'm not I am declared!";
  </script>
</head>
<body>
</body>
</html>
```



and



. In the console, you no longer see the error, and you can type "myVar" and see its value.

You can use "use strict" at the global level, as well as inside individual functions. If you put "use strict" at the global level, it affects *all* your code. If you put it inside a function only, it will affect just the code in that function. So you could put all your strict code into an IIFE, like this:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    "use strict";
    var myVar = "I am declared!";
    (function() {
      "use strict";
      innerMyVar = "I'm not declared in this function either.";
    }) ();

  </script>
</head>
<body>
</body>
</html>
```



and **Preview**. In the console, you'll see a reference error. You can fix it by adding "var" in front of the variable **innerMyVar**.

Let's look at one other way that strict mode can help prevent mistakes:

#### CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    (function() {
      "use strict";
      var innerMyVar = "I'm not declared in this function either.";

      var o = {
        x: 3,
        x: 10
      };
    }) ();
  </script>
</head>
<body>
</body>
</html>
```

We added an object, **o**, and we defined the property **o.x** *twice* in the object definition. Let's see what happens in strict mode. and **Preview**. In the console, you see an error (in Chrome):

#### OBSERVE:

```
Uncaught SyntaxError: Duplicate data property in object literal not allowed in s
trict mode
```

Previously, it was perfectly allowable to define the same property twice in an object (although not a good idea). Try it and see what happens (you can try it by commenting out the **"use strict"**; line with //). In strict mode, we can't do this anymore, and the browser generates an error that describes the problem.

Strict mode can help you write cleaner and better code. Strict mode will help you with other tasks as well; check out the [MDN Developer Network's strict mode page](#) for more.

The way we tell the browser to use strict mode might seem a little odd; after all, it's just a string, "use strict."

Why do you think the language designers decided to do it this way?

They did it this way because so that older browsers that don't support strict mode could safely ignore the statement. To older browsers, "use strict" is simply a string and won't affect how your code runs at all. To newer browsers, of course, the string causes the browser to go into strict mode. This means you can use strict mode in your code without worrying that older browsers won't be able to run the code. However, pay particular attention if you are linking to multiple scripts in your page. You can combine scripts if they are all strict, or all non-strict, but you can't mix the two! Make sure you know the mode of all of your scripts use before linking to them.

## New Methods

There are a few new methods that have been added to objects in the language. Let's check out some of the most useful of them.

For strings, the **trim()** method is now part of the language. Browsers have implemented this for a while. The useful **trim()** method removes white space at the beginning and ending of a string:

### INTERACTIVE SESSION:

```
> var s = "  Lots of white space here  ";
undefined
> s
"  Lots of white space here  "
> s.trim()
"Lots of white space here"
```

Nice! It's possible to implement a **trim()** function yourself, as you saw in an earlier project, but it's so much nicer to have it built into the language.

There are some new array methods to o:

### INTERACTIVE SESSION:

```
> var a = [1, 2, 3];
undefined
> a
[1, 2, 3]
> a.forEach(function(x, i, a) { a[i] = x + 1; });
undefined
> a
[2, 3, 4]
```

Here, we define an array **a**, and then use the **forEach()** method to apply a function to each element of the array. (This may seem familiar to you, since you've already implemented your own **forEach()** function earlier in the course.) **forEach()** takes a function, and applies that function to each element of the array to change it. The three parameters of the function are **x** (the value of the current array item to which the function is being applied), **i** (the index of the current array item), and **a** (the array itself). Our function adds 1 to each item in the array. Notice that the **forEach** method doesn't return anything, but (in this case) our function modifies the array directly.

Now let's try the **map()** method (you've seen this before):

### INTERACTIVE SESSION:

```
> var b = a.map(function(x) { return x * 2; });
undefined
> b
[4, 6, 8]
> a
[2, 3, 4]
```

**map()** applies a function to each item in the array, and creates a new array out of the return values from the function. Here, we multiply each item in **a** by 2, and store the resulting values in a new array named **b**. Notice that **a** doesn't change.

ES5 defines a few other useful array methods you should check out, including **isArray()**, **every()**, **some()**, **filter()** and **reduce()**.

## Object Property Descriptors

In ES5, objects are quite a bit more complex. Fortunately, you don't have to change the way you create and use objects. These new features are useful, but definitely not required.

The big change in objects is that an object property now comes with a *property descriptor*. In fact, assuming you're using a browser that implements the ES5 standard for JavaScript, the objects you've been creating in this course all have properties with property descriptors, you just didn't know it. You already know that an object property has a value. In addition, properties now have three other attributes: **writable**, **enumerable**, and **configurable**. These three attributes are all optional, and are all true by default. Let's check them out:

### INTERACTIVE SESSION:

```
> var o = { x: 1 }
undefined
> Object.getOwnPropertyDescriptor(o, "x")
Object { value: 1, writable: true, enumerable: true, configurable: true }
```

We define a simple object **o**, with just one property **o.x** which has the value 1.

Then we use a method of **Object** (which, remember, is the parent object of all other objects in JavaScript), **Object.getOwnPropertyDescriptor()** to get the property descriptor for the property **x**. We pass in both the object and the name of the property (as a string) as arguments to the method, and get back the property descriptor.

The property descriptor shows the value of the property, 1, as well as the values for the attributes **writable**, **enumerable**, and **configurable**. Let's talk about these attributes.

If **writable** is **true**, then you can change the value of an attribute:

### INTERACTIVE SESSION:

```
> o.x = 10
10
> o
Object { x: 10 }
```

Just like you'd expect, that's the default behavior. So, what if you want to change it? There are a couple of ways you can do that. One is to modify the attributes of the property descriptor directly using an **Object** method, **Object.defineProperty()**. We'll try this approach now, but there's a simpler way to change an object to prevent its properties from being writable that we'll look at a bit later.

### INTERACTIVE SESSION:

```
> Object.defineProperty(o, "x", { writable: false });
Object { x: 10 }
> o.x = 20;
20
> o.x
10
```



The **Object.defineProperty()** method takes three arguments: the object whose property you want to modify, the property name (and note that this is a string), and an object with the attribute you want to modify. In this case, we've passed an object setting the **writable** attribute to **false**. We then try to change the value of



the property `o.x` to 20, and we can't; the value of `o.x` is still 10. Note that we don't get an error! The assignment is simply ignored. That's because we're not using strict mode in the console. Let's try the same operation using strict mode. Modify `strict.xml` as shown:

```
CODE TO TYPE:
<!doctype html>
<html>
<head>
  <title> </title>
  <meta charset="utf-8">
  <script>
    (function() {
      "use strict";
      var innerMyVar = "I'm not declared in this function either.";

      var o = {
        x: 37
        x: 10
      };
      Object.defineProperty(o, "x", { writable: false });
      o.x = 20;
    })();
  </script>
</head>
<body>
</body>
</html>
```

Here, we define an object `o`, change the property `x` so that it's not writable, and then attempt to set the value of `x` to 20.  and . In the console, you see this error (in Chrome):

```
OBSERVE:
Uncaught TypeError: Cannot assign to read only property 'x' of #<Object>
```

So, in strict mode, attempting to set the value of a property that's not writable will generate an error; if you're not in strict mode, the assignment will fail, but you won't get an error (and your code will continue to execute). We'll continue to work in the console, but we'll note where an operation that's ignored in the console would generate an error in strict mode.

Keep in mind that when you modify an attribute of a property using `Object.defineProperty()`, you're not modifying an attribute of the *entire object*; you're modifying only an attribute of a *specific property* in the object. Every property that you add to the object will have its own set of attributes. Let's try adding a new property to `o`:

```
INTERACTIVE SESSION:
> var o = { x: 10 }
undefined
> Object.defineProperty(o, "x", { writable: false });
Object {x: 10}
> o.y = 20;
20
> o
Object {x: 10, y: 20}
> Object.getOwnPropertyDescriptor(o, "x");
Object { value: 10, writable: false, enumerable: true, configurable: true }
> Object.getOwnPropertyDescriptor(o, "y");
Object { value: 20, writable: true, enumerable: true, configurable: true }
```

First we add a new property to `o`; that works as usual. Then we display the property descriptors for the two properties. You can see that `o.x` is not writable, while `o.y` is (because `writable` is `true` by default).

Okay, we've explored the `writable` attribute, now what about `enumerable`? This property indicates that you can *enumerate* the property when you loop through the object's properties (or use one of the new Object methods to display properties). Let's give it a try:

#### INTERACTIVE SESSION:

```
> o
Object { x: 10, y: 20 }
> for (var prop in o) { console.log(prop); }
x
y
< undefined
```

Our object `o` has two properties and both are currently enumerable (take a look back at the `enumerable` attribute for both properties above, and you'll see that both attribute values are `true`). That means when we enumerate them by looping through all the property names in the object using the `for` loop above, we can see both property names.

ES5 has added two other methods for enumerating object property names: `Object.keys()` and `Object.getOwnPropertyNames()`. Let's try both of these methods to enumerate the property names in `o`:

#### INTERACTIVE SESSION:

```
> Object.keys(o);
["x", "y"]
> Object.getOwnPropertyNames(o);
["x", "y"]
```

In both cases, the result is an array of property names. These new methods for retrieving the property names of an object (added in ES5) are really efficient. When you use them, you don't have to write a loop to access each property name.

Now, let's try making the `o.y` property's `enumerable` attribute `false`:

#### INTERACTIVE SESSION:

```
> Object.defineProperty(o, "y", { enumerable: false });
Object { x: 10 }
> o
Object { x: 10 }
> Object.getOwnPropertyDescriptor(o, "y");
Object { value: 20, writable: true, enumerable: false, configurable: true }
> Object.keys(o);
["x"]
> Object.getOwnPropertyNames(o);
["x", "y"]
```

First, we set the `enumerable` attribute of `o.y` to `false`. You can see right away that when we display the object, we no longer see the `y` property. That means the `Object.toString()` method can't "see" properties that aren't enumerable.

We get the property descriptor for `o.y` and see that indeed, the `enumerable` attribute has been set to `false`. When we use the `Object.keys()` method to retrieve the property names in the object `o`, we no longer see "y" in the resulting array. So the `Object.keys()` method can't "see" properties that aren't enumerable either.

Finally, we use the `Object.getOwnPropertyNames()` method to get the property names and again we see "y" in the results.

So, we have a way of hiding property names when an object is displayed, or when we try to enumerate the property names with any method except `Object.getOwnPropertyNames()`.

Now, let's look at the **configurable** attribute. Once you set this attribute to `false`, you can't change it back because the property is no longer configurable. If a property isn't configurable, it can't be deleted from the object.

#### INTERACTIVE SESSION:

```
> o.z = 3;
3
> o
Object {x: 10, z: 3}
> Object.defineProperty(o, "z", { configurable: false });
Object {x: 10, z: 3}
> o.z = 4;
4
> o
Object {x: 10, z: 4}
> delete o.z
false
> Object.defineProperty(o, "z", { configurable: true });
TypeError: Cannot redefine property: z
```

First, we add a new property, `o.z` and set its value to 3. Then we change the `o.z` property's **configurable** attribute to `false`. We can still change the value of the property, which we do, setting it to 4, but we can't delete the property, or set its **configurable** attribute back to `true`.

**Note** Attempting to delete a non-configurable property in strict mode generates the error: **Uncaught TypeError: Cannot delete property 'z' of #<Object>**

These attributes allow you to have a lot more control over your objects' properties.

## Sealing and Freezing Objects

There are two shortcut methods you can use to set property descriptor attributes and help protect your objects.

The first of these is `Object.seal()`. This method sets the **configurable** attribute of every property in the object to `false`, and also disallows the addition of any new properties to the object. You can still read, write, and enumerate the properties in the object; you just can't remove properties or add new ones.

## INTERACTIVE SESSION:

```
> var myObject = {
    name: "Elisabeth",
    course: "Advanced JavaScript",
    year: 2013
};
undefined
> myObject
Object {name: "Elisabeth", course: "Advanced JavaScript", year: 2013}

> Object.getOwnPropertyDescriptor(myObject, "name");
Object {value: "Elisabeth", writable: true, enumerable: true, configurable: true}
> Object.getOwnPropertyDescriptor(myObject, "course");
Object {value: "Advanced JavaScript", writable: true, enumerable: true, configurable: true}
> Object.getOwnPropertyDescriptor(myObject, "year");
Object {value: 2013, writable: true, enumerable: true, configurable: true}

> Object.seal(myObject);
Object {name: "Elisabeth", course: "Advanced JavaScript", year: 2013}

> Object.getOwnPropertyDescriptor(myObject, "name");
Object {value: "Elisabeth", writable: true, enumerable: true, configurable: false}
> Object.getOwnPropertyDescriptor(myObject, "course");
Object {value: "Advanced JavaScript", writable: true, enumerable: true, configurable: false}
> Object.getOwnPropertyDescriptor(myObject, "year");
Object {value: 2013, writable: true, enumerable: true, configurable: false}

> myObject.newProperty = "attempting to add a new property";
"attempting to add a new property"
> myObject
Object {name: "Elisabeth", course: "Advanced JavaScript", year: 2013}
> Object.keys(myObject);
["name", "course", "year"]
> myObject.name = "Scott";
Object {name: "Scott", course: "Advanced JavaScript", year: 2013}
```

First, we define a new object, **myObject**, with three properties: **name**, **course**, and **year**. By default, the attributes in the property descriptor for each of these properties are **true**, which we can see by inspecting them using **Object.getOwnPropertyDescriptor()**. (Unfortunately, there's no way to see the property descriptors of all the properties at once).

Next, we "seal" the object, by calling **Object.seal()**, and passing in the object **myObject**. This changes the **configurable** attribute of each of the properties to **false**.

We try to add a new property to the object, **myObject.newProperty**. We don't get an error message when we try to do this (because we're not in strict mode), but the assignment is ignored, and the property is not added to the object. When we use the **Object.keys()** method to display the property names in the object, we can see that no new property is added.

Finally, we change the value of the **myObject.name** property from "Elisabeth" to "Scott," just to prove that even if the object is sealed, we can still change the value of the properties.

**Note** If you try to add a new property to **myObject** (which is sealed) in strict mode, you'll get the error message **Uncaught TypeError: Can't add property newProperty, object is not extensible**.

You can check to see if an object is sealed using the **Object.isSealed()** method:

#### INTERACTIVE SESSION:

```
> Object.isSealed(myObject)
true
```

A similar method is **Object.freeze()**. It works like **Object.seal()**, but in addition, it sets the **writable** attribute of all the property descriptors to **false**, so you can't change the value of properties:

#### INTERACTIVE SESSION:

```
> Object.freeze(myObject);
Object {name: "Scott", course: "Advanced JavaScript", year: 2013}
> myObject.name = "Elisabeth"
"Elisabeth"
> myObject
Object {name: "Scott", course: "Advanced JavaScript", year: 2013}
```

You can find out if an object is frozen using the **Object.isFrozen()** method.

#### Note

If you try to set the value of a property in a frozen object in strict mode, you'll get the error message **Uncaught TypeError: Cannot assign to read only property 'name' of #<Object>**.

The **Object.seal()** and **Object.freeze()** methods are useful for protecting your objects while still allowing access to the properties (both for reading the value of the properties and for enumerating the properties).

Both **Object.seal()** and **Object.freeze()** disallow new properties from being added to objects by setting them to *non-extensible*. You can do this yourself (separately) using the **Object.preventExtensions()** method, and determine whether an object is extensible using the **Object.isExtensible()** method.

We left the description of these recent capabilities of JavaScript objects until the end of the course because they are recent additions and as such are not used much yet. Still, it's important to know that they are available if and when you do need them. You probably will in certain situations like when you want to make sure that an object is protected from change. In addition, it's likely that these features will be used in future additions to the language.

## Creating Objects

Another new method of **Object** that was added in ES5 is the **Object.create()**. You already know how to create objects by writing a literal object, and by using a constructor.

**Object.create()** is a little different because it allows you to create an object and specify its prototype. So if you create an object, like the **Person** object below, you can then create objects that use **Person** as their prototype (meaning those objects inherit the properties and methods of the **Person** object):

## CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title> Creating Objects </title>
  <meta charset="utf-8">
  <script>
    var Person = {
      welcome: function() {
        console.log("Welcome " + this.name + "!");
      },
      isAdult: function() {
        if (this.age > 17) {
          return true;
        }
      }
    };

    var bob = Object.create(Person);
    bob.name = "Bob Parsons";
    bob.age = 42;

    bob.welcome();
    if (bob.isAdult()) {
      console.log(bob.name + " can get a beer");
    }

    var mary = Object.create(Person);
    mary.name = "Mary Smith";
    mary.age = 12;

    mary.welcome();
    if (!mary.isAdult()) {
      console.log("Sorry, " + mary.name + " can't get a beer");
    }
  </script>
</head>
<body>
</body>
</html>
```



Save the file in your **/AdvJS/** folder as **createObjects.html** and **Preview** . Open the console; you see:

## OBSERVE:

```
Welcome Bob Parsons!
Bob Parsons can get a beer
Welcome Mary Smith!
Sorry, Mary Smith can't get a beer
```

We made these variables global so you can access them in the console. Try this:

#### INTERACTIVE SESSION:

```
> Person.isPrototypeOf(bob)
true
> Person.isPrototypeOf(mary)
true
> bob
Object {name: "Bob Parsons", age: 42, welcome: function, isAdult: function}
> mary
Object {name: "Mary Smith", age: 12, welcome: function, isAdult: function}
```

The **Person** object is the prototype of both **bob** and **mary**, and those objects do indeed inherit the properties from **Person**. (So, what happens when you use **Object.keys()** to get the property names of **bob** and **mary**? Why?)

However, note that because we created **bob** and **mary** using **Object.create()** and not a constructor, when you look at the constructor of **bob** like this:

#### INTERACTIVE SESSION:

```
> bob.constructor
function Object() { [native code] }
> bob
Object {name: "Bob Parsons", age: 42, welcome: function, isAdult: function}
```

...you can see that the constructor is **Object()**. **bob** and **mary** are created using the **Object()** constructor behind the scenes and then explicitly changing the prototype object to **Person**. You can see this reflected also when we display **bob** in the console: you see "Object" as the "type" of the object, even though the prototype of the object is **Person**.

Whenever you create an object using a constructor, that object gets a prototype that is stored in the constructor's **prototype** property. Recall that we created a circle by writing **new Circle()**, and the resulting object's prototype was the **Circle** object in **Circle.prototype**. Contrast that way of creating objects with using **Object.create()**. **Object.create()** allows you to assign the prototype of an object without (explicitly) using a constructor (although the **Object()** constructor is used behind the scenes).

In addition, with **Object.create()** you can pass a second argument to the method containing an object that specifies attributes for the properties:

## INTERACTIVE SESSION:

```
> var jim = Object.create(Person, {
  name: {
    value: "Jim Smith",
    writable: false
  },
  age: {
    value: 42,
    writable: false
  }
});
undefined
> jim.name
"Jim Smith"
> jim.age
42
> jim.welcome()
Welcome Jim Smith!
< undefined
> jim.isAdult()
true
> jim.name = "Joe Schmoe"
"Joe Schmoe"
> jim.welcome()
Welcome Jim Smith!
```

Here, we create a new object using **Object.create()**, and pass the **Person** object to use as the prototype. We also pass an object that defines the values of the properties and their attributes for the new object. Because we set both properties so they are not writable, we can't change the values in the object **jim**.

**Note** If you try to change the value of **jim.name** in strict mode, you see the error message **Uncaught TypeError: Cannot assign to read only property 'name' of #<Object>**.

So, **Object.create()** gives you an efficient way to create a prototype chain for objects in JavaScript.

In this lesson, you've learned about some of the new features of JavaScript that were added in ES5 (or ECMAScript 5.1 specification). If you're running the most recent version of a modern browser, then it's likely you can use all of these new features.

Developers are already hard at work on the ECMAScript 6 specification for the next version of JavaScript (code-named Harmony). You can see the specification in progress at the [ECMAScript wiki](#). As of this writing, the creation of the new specification is underway, so many of the new features are not available in browsers. The changes will be more extensive than the changes in ES5, including the addition of block scope, classes, and modules. Exciting!

You've come a long way in this course, exploring much of the JavaScript language including types and values, functions, objects, and closures. You've also worked with techniques for programming such as using the Module pattern to reduce global variables and keep parts of objects private. Well done! We look forward to seeing what you do in your final project!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.