Riverside JS Workshop
JavaScript Skills Ladder, Lesson 3.1

Introduction to Advanced JavaScript

Welcome to of the Core JavaScript Skills Ladder, Phase 3! In this phase, you'll learn advanced JavaScript that will let you build sophisticated JavaScript applications.

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.*

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. These Skills Ladder lessons are 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.

Here are some tips for using these lessons 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.
- **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.

- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes!

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 work folder for your code files. Now create a new HTML file and 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 work folder. Now create a new JavaScript file and add this code:

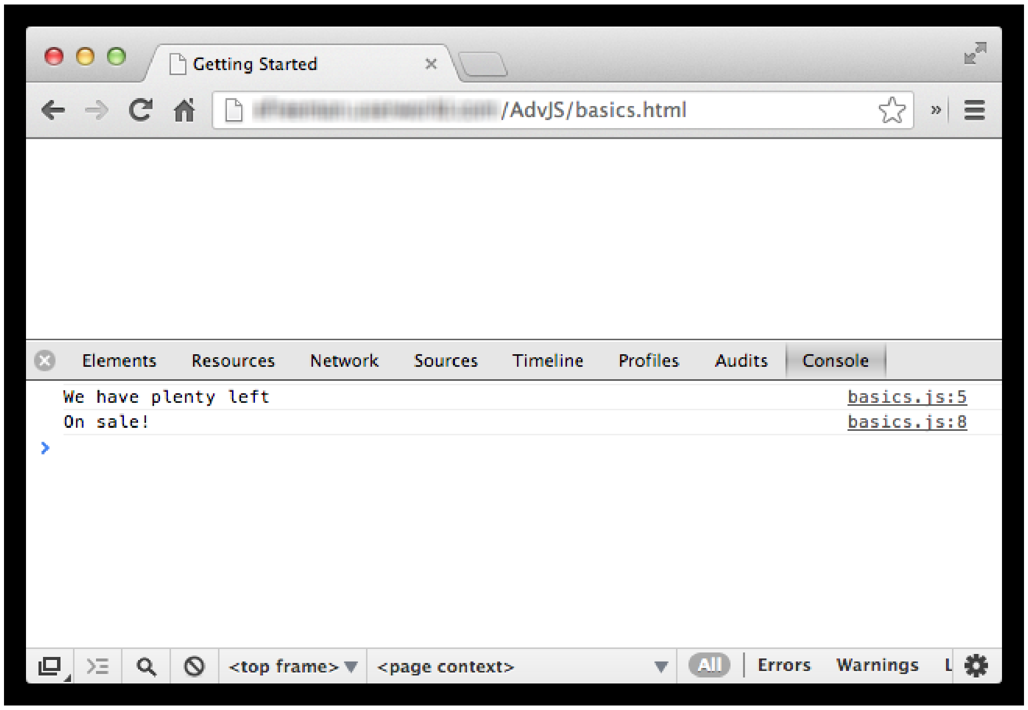
CODE TO TYPE:

```
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 work folder, and open *basics.html* in the browser. You'll see an empty web page. To see the result of the JavaScript, open up your browser's console. Because browsers are constantly changing, if you need help finding the console in your browser search “open JavaScript console in *browser_name.*”

Using the Console

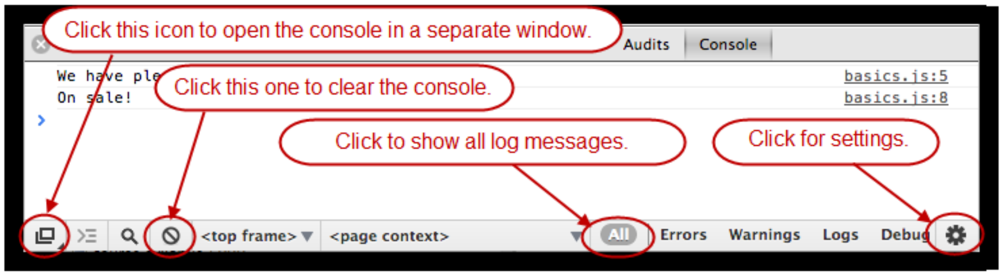
Let's take a closer look at the Chrome console since that's the one we'll use in our examples throughout the course. Other browser consoles have the same basic functionality. We'll let you know when you need to use the Chrome console specifically to follow along.


Getting Started
/AdvJS/basics.html
Elements Resources Network Sources Timeline Profiles Audits Console
<top frame> <page context>
All Errors Warnings

```
We have plenty left                                   basics.js:5
On sale!                                              basics.js:8
```

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. 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:

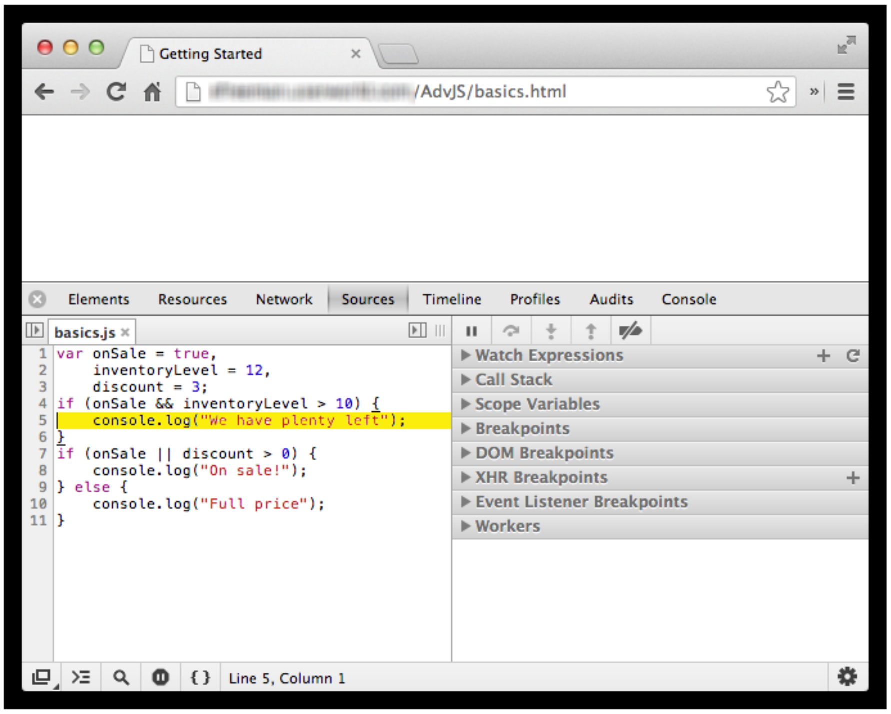

Click this icon to open the console in a separate window.
Click this one to clear the console.
Click to show all log messages.
Click for settings.
Audits Console
<top frame> <page context>
All Errors Warnings Logs Debug

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 is 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:



```
1  var onSale = true,
2      inventoryLevel = 12,
3      discount = 3;
4  if (onSale && inventoryLevel > 10) {
5      console.log("We have plenty left");
6  }
7  if (onSale || discount > 0) {
8      console.log("On sale!");
9  } else {
10     console.log("Full price");
11 }
```

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 43. 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. Just search for online for code compression tools.

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).

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:

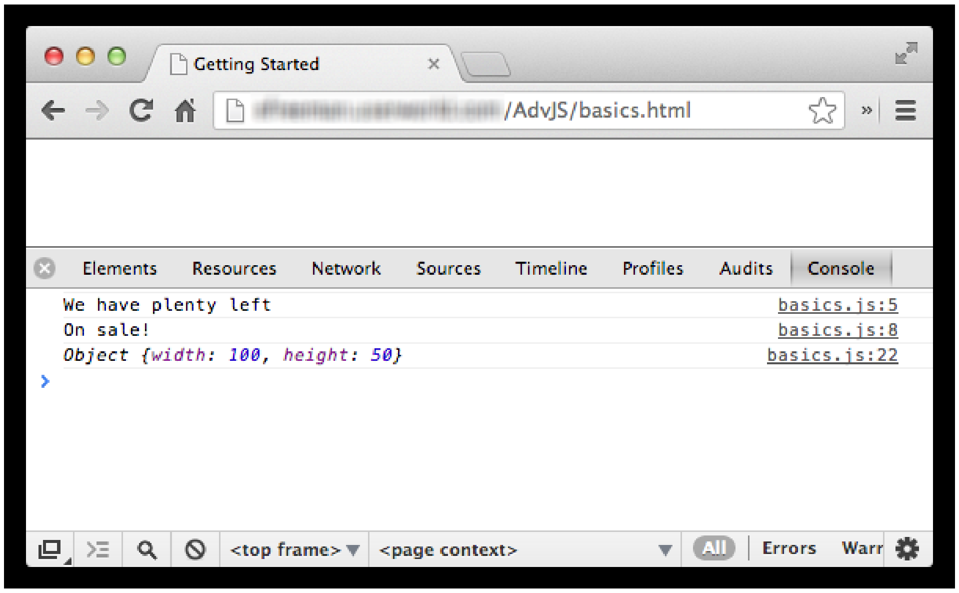
```
var onSale = true,
    inventoryLevel = 12,
    discount = 3;
if (onSale && inventoryLevel > 10) {
    console.log("We have plenty left");
}
if (onSale || discount > 0) {
    console.log("On sale!");
} else {
    console.log("Full price");
}
var rect = {
    width: 100,
    height: 50
};
console.log(rect);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

Save your changes to *basics.js*, and *basics.html* in your 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:

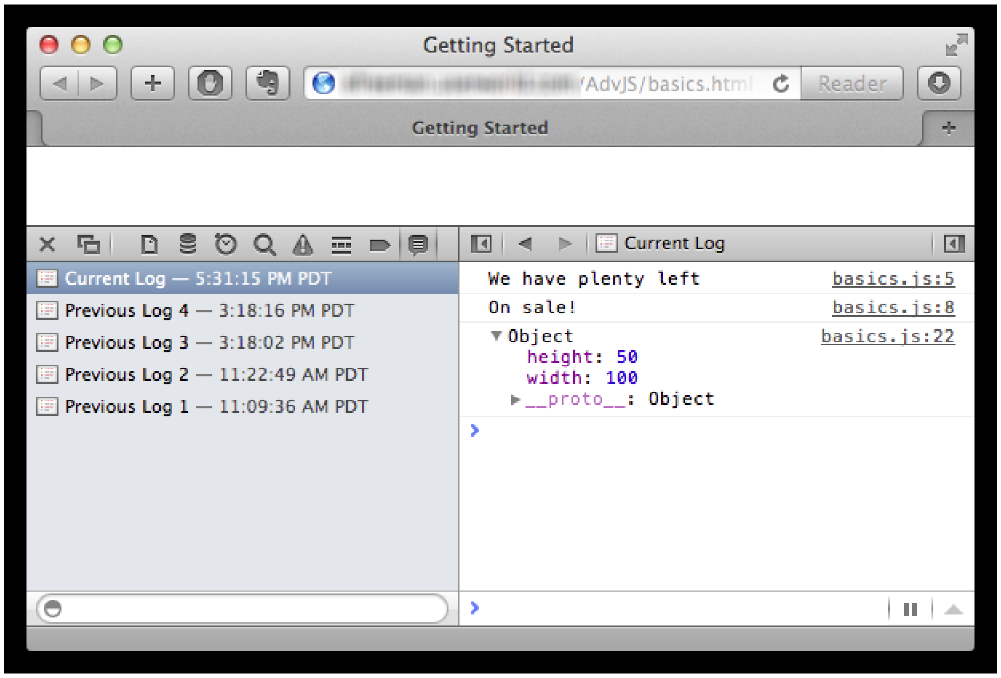
```
We have plenty left
On sale!
Object { width: 100, height: 50 }
```

In Chrome, it will look like this:



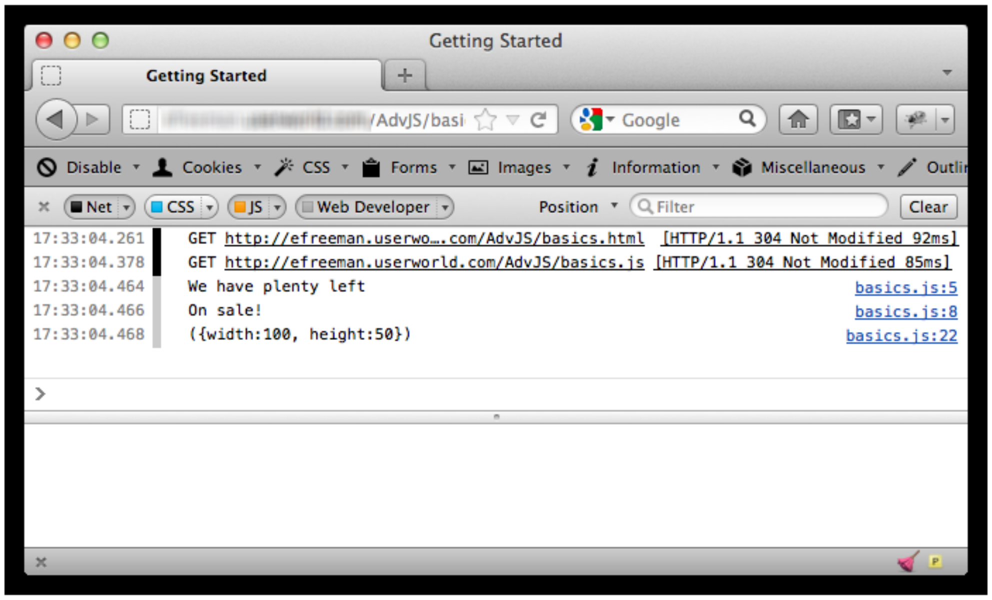
```
We have plenty left                                   basics.js:5
On sale!                                              basics.js:8
Object {width: 100, height: 50}                       basics.js:22
```

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


Getting Started
/AdvJS/basics.html
Reader
Getting Started
Current Log
Current Log — 5:31:15 PM PDT
Previous Log 4 — 3:18:16 PM PDT
Previous Log 3 — 3:18:02 PM PDT
Previous Log 2 — 11:22:49 AM PDT
Previous Log 1 — 11:09:36 AM PDT

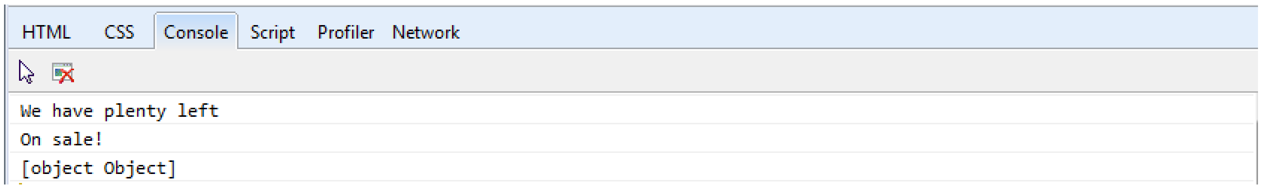
```
We have plenty left             basics.js:5
On sale!                        basics.js:8
▼Object                         basics.js:22
    height: 50
    width: 100
  ▶__proto__: Object
```

In Firefox, it will look like this:



```
17:33:04.261  GET http://efreeman.userwo….com/AdvJS/basics.html  [HTTP/1.1 304 Not Modified 92ms]
17:33:04.378  GET http://efreeman.userworld.com/AdvJS/basics.js [HTTP/1.1 304 Not Modified 85ms]
17:33:04.464  We have plenty left                                              basics.js:5
17:33:04.466  On sale!                                                         basics.js:8
17:33:04.468  ({width:100, height:50})                                        basics.js:22
```

In IE, it will look like this:



```
We have plenty left
On sale!
[object Object]
```

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);
```

Save your changes to *basics.js*, and *basics.html* in your browser. 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. Save your changes to *basics.js*, and *basics.html* in your browser.

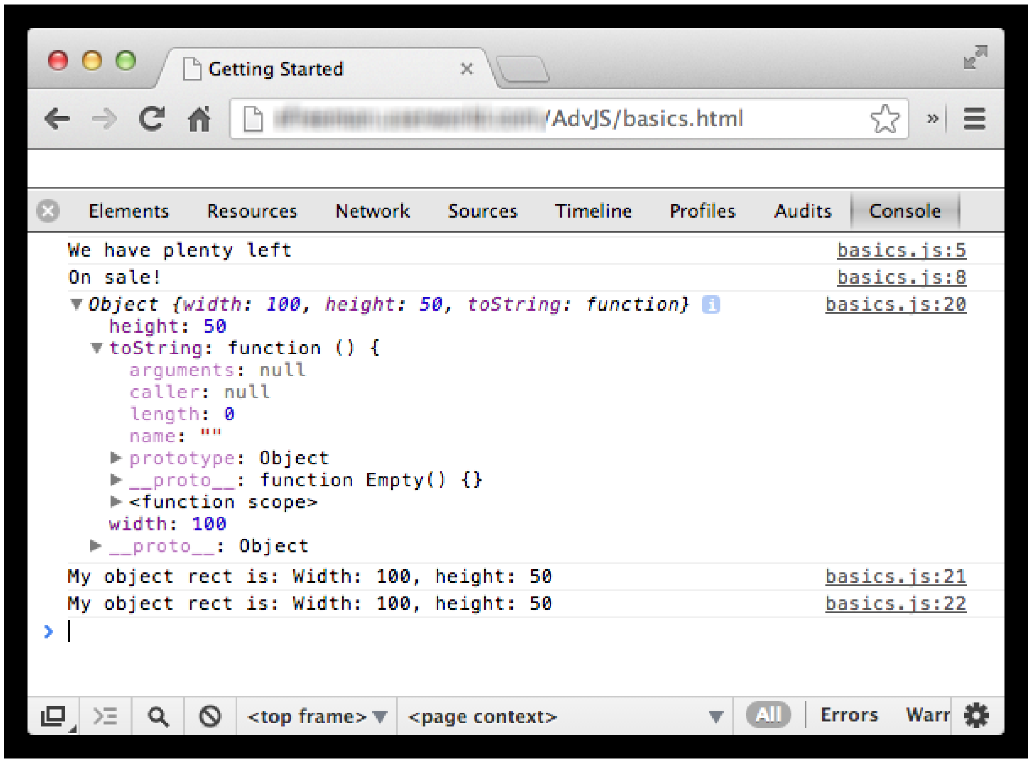
OBSERVE:

```
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.

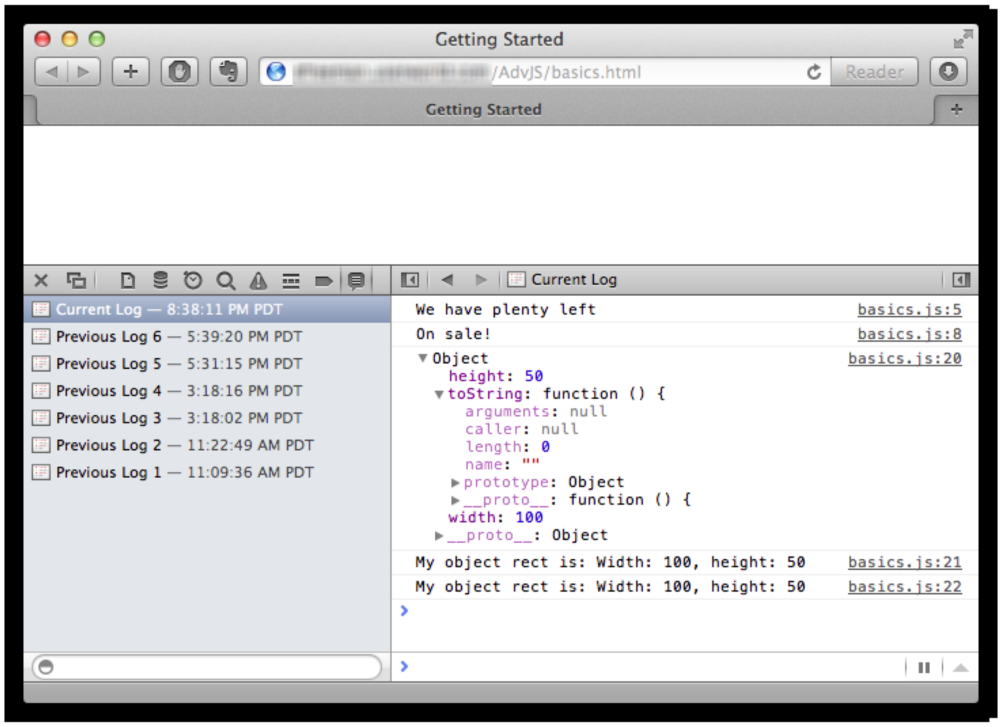
Don't worry about these details right now. We'll come back to all that later. For now just note the different ways that the console displays output, depending on how you call `console.log()` and the kind of value you pass this function.

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.



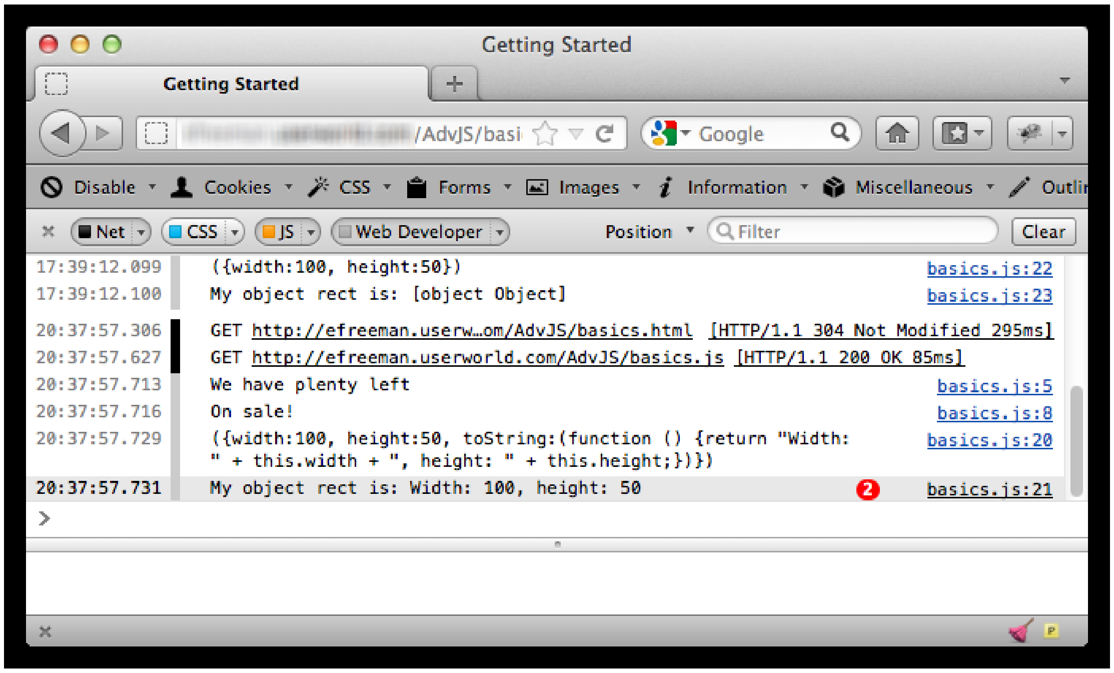
```
We have plenty left                                        basics.js:5
On sale!                                                   basics.js:8
▼Object {width: 100, height: 50, toString: function}      basics.js:20
   height: 50
  ▼toString: function () {
     arguments: null
     caller: null
     length: 0
     name: ""
    ▶prototype: Object
    ▶__proto__: function Empty() {}
    ▶<function scope>
   width: 100
  ▶__proto__: Object
My object rect is: Width: 100, height: 50                 basics.js:21
My object rect is: Width: 100, height: 50                 basics.js:22
> |
```

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


Getting Started
[redacted]/AdvJS/basics.html
Reader
Getting Started
Current Log
Current Log — 8:38:11 PM PDT
Previous Log 6 — 5:39:20 PM PDT
Previous Log 5 — 5:31:15 PM PDT
Previous Log 4 — 3:18:16 PM PDT
Previous Log 3 — 3:18:02 PM PDT
Previous Log 2 — 11:22:49 AM PDT
Previous Log 1 — 11:09:36 AM PDT

```
We have plenty left                               basics.js:5
On sale!                                          basics.js:8
▼Object                                          basics.js:20
   height: 50
  ▼toString: function () {
     arguments: null
     caller: null
     length: 0
     name: ""
    ▶prototype: Object
    ▶__proto__: function () {
   width: 100
  ▶__proto__: Object
My object rect is: Width: 100, height: 50        basics.js:21
My object rect is: Width: 100, height: 50        basics.js:22
>
```

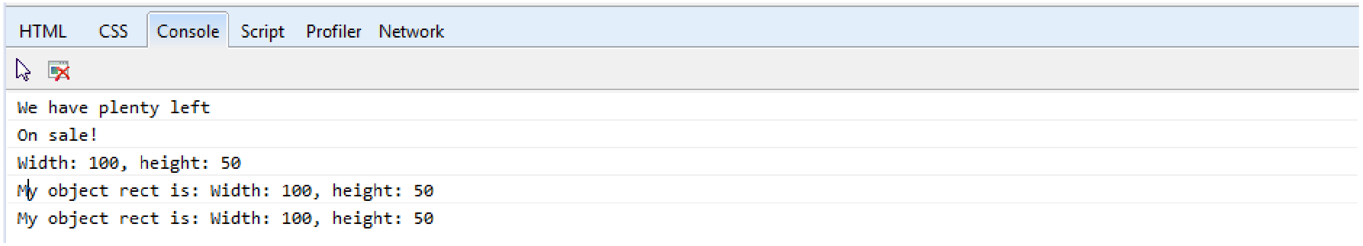
Firefox:



```
17:39:12.099    ({width:100, height:50})                                          basics.js:22
17:39:12.100    My object rect is: [object Object]                                basics.js:23
20:37:57.306    GET http://efreeman.userw…om/AdvJS/basics.html  [HTTP/1.1 304 Not Modified 295ms]
20:37:57.627    GET http://efreeman.userworld.com/AdvJS/basics.js  [HTTP/1.1 200 OK 85ms]
20:37:57.713    We have plenty left                                                basics.js:5
20:37:57.716    On sale!                                                           basics.js:8
20:37:57.729    ({width:100, height:50, toString:(function () {return "Width:     basics.js:20
                " + this.width + ", height: " + this.height;})})
20:37:57.731    My object rect is: Width: 100, height: 50              2          basics.js:21
>
```

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

IE:



```
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.

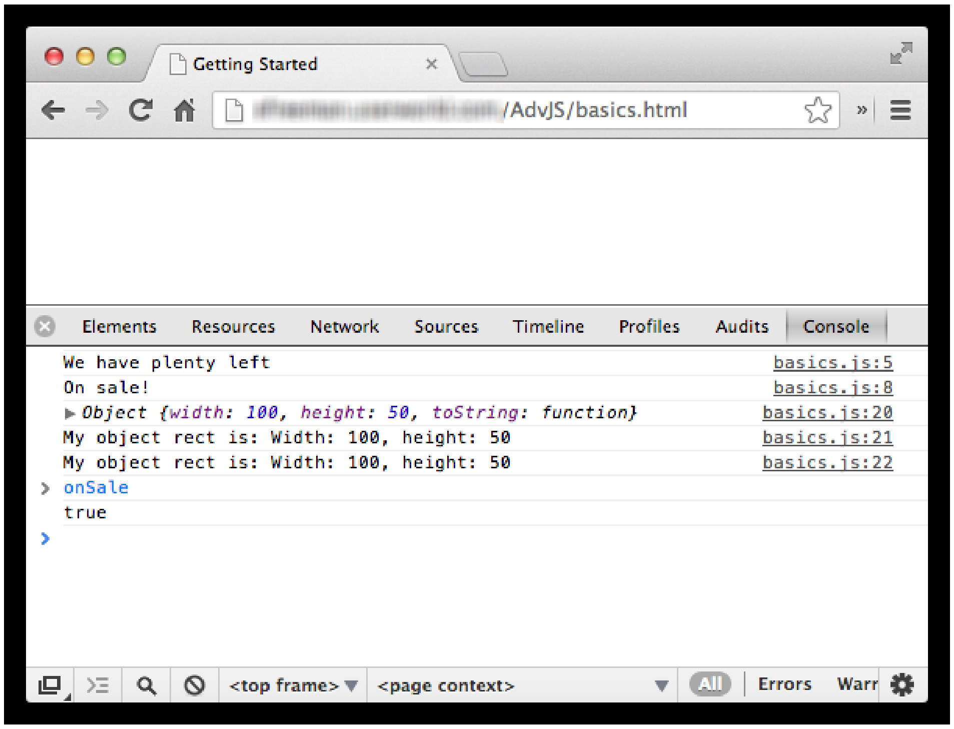
In your browser's console, you'll see a prompt which indicates where you can type your own JavaScript. Click in the console window next to the prompt, and type this:

INTERACTIVE SESSION:

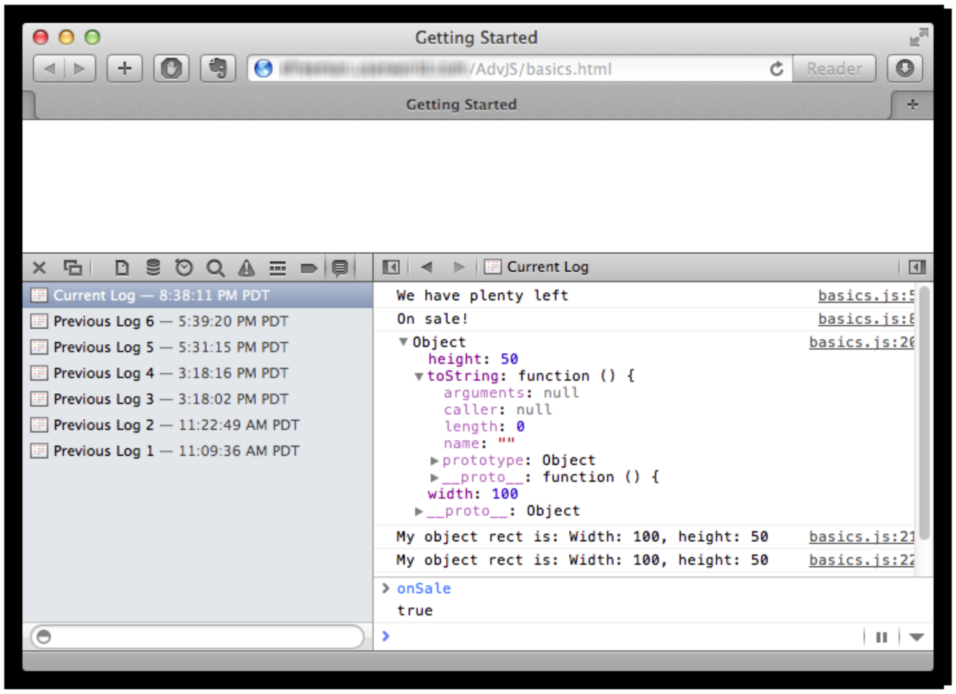
```
> onSale
  true
```

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

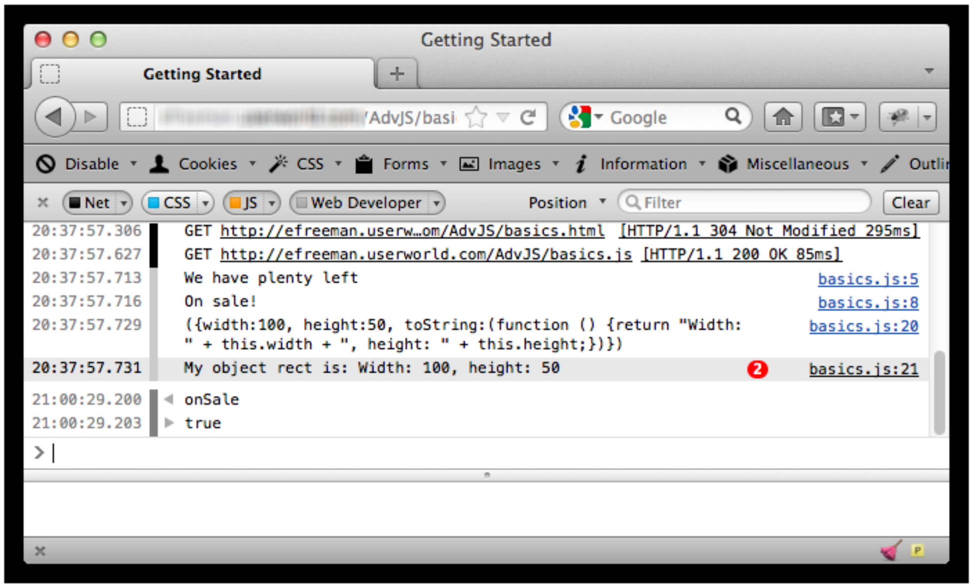
We typed the name of the global variable `onSale` and pressed *Enter*. JavaScript responded with the value of `onSale`. Here's what the output looks like in the browsers Chrome, Safari, Firefox, and IE:



```
We have plenty left                                      basics.js:5
On sale!                                                 basics.js:8
▶Object {width: 100, height: 50, toString: function}    basics.js:20
My object rect is: Width: 100, height: 50               basics.js:21
My object rect is: Width: 100, height: 50               basics.js:22
> onSale
  true
>
```

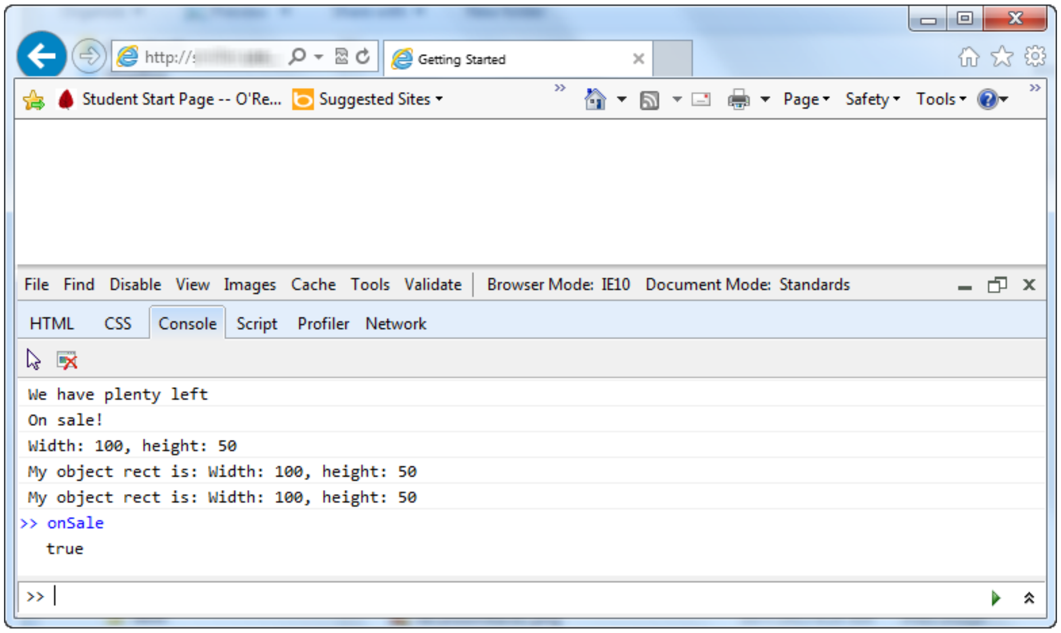

Getting Started
[redacted]/AdvJS/basics.html
Reader
Getting Started
Current Log
Current Log — 8:38:11 PM PDT
Previous Log 6 — 5:39:20 PM PDT
Previous Log 5 — 5:31:15 PM PDT
Previous Log 4 — 3:18:16 PM PDT
Previous Log 3 — 3:18:02 PM PDT
Previous Log 2 — 11:22:49 AM PDT
Previous Log 1 — 11:09:36 AM PDT

```
We have plenty left                              basics.js:5
On sale!                                         basics.js:8
▼Object                                          basics.js:20
    height: 50
  ▼toString: function () {
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶prototype: Object
    ▶__proto__: function () {
    width: 100
  ▶__proto__: Object
My object rect is: Width: 100, height: 50        basics.js:21
My object rect is: Width: 100, height: 50        basics.js:22
> onSale
  true
>
```



```
20:37:57.306  GET http://efreeman.userw…om/AdvJS/basics.html [HTTP/1.1 304 Not Modified 295ms]
20:37:57.627  GET http://efreeman.userworld.com/AdvJS/basics.js [HTTP/1.1 200 OK 85ms]
20:37:57.713  We have plenty left                                        basics.js:5
20:37:57.716  On sale!                                                   basics.js:8
20:37:57.729  ({width:100, height:50, toString:(function () {return "Width:   basics.js:20
              " + this.width + ", height: " + this.height;})})
20:37:57.731  My object rect is: Width: 100, height: 50            2      basics.js:21

21:00:29.200  ◀ onSale
21:00:29.203  ▶ true
> |
```



File Find Disable View Images Cache Tools Validate Browser Mode: IE10 Document Mode: Standards
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
>> onSale
  true
>>
```

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 (`inventory`, `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 * Math.PI * 2;
}
console.log("Area is: " + computeArea(3));
```

Here, we used /* ... */ to comment out large chunks of code. This is standard practice, as the /* and */ delimeters 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.

More about the material in this lesson

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