Enterprise Web Development

Yakov Fain, Victor Rasputnis, Anatole Tartakovsky, and Viktor Gamov

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Enterprise Web Development

by Yakov Fain, Victor Rasputnis, Anatole Tartakovsky, and Viktor Gamov

Printed in the United States of America.

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

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

Editors: Mary Treseler and Brian Anderson
Production Editor: Melanie Yarbrough
Copyeditor: Sharon Wilkey
Proofreader: Kim Cofer
Indexer: Judith McConville
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

June 2014: First Edition

Revision History for the First Edition:

2014-06-25: First release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449356811* for release details.

ISBN: 978-1-449-35681-1

[LSI]

Table of Contents

A. Advanced Introduction to JavaScript. 5

APPENDIX B

Advanced Introduction to JavaScript

> Any application that can be written in JavaScript, will eventually be written in JavaScript.
>
> — Atwood's Law

This appendix is dedicated to the JavaScript programming language. Although all chapters of this book show how JavaScript frameworks can greatly minimize the amount of JavaScript code that you need to write manually, you still need to understand the language itself. We assume that you have some programming experience, understand HTML syntax, and are familiar with the general principal of communication between web browsers and web servers. We've included the word *advanced* in the title of this appendix because of these assumptions. We'll begin by covering basics of the language, but then quickly progress to such advanced topics as *prototypal inheritance*, *callbacks*, and *closures*.

If you're an absolute beginner with web development and have no previous exposure to JavaScript, consider reading one of the fundamental tutorials covering each and every detail of JavaScript. We can recommend *JavaScript: The Definitive Guide*, by David Flanagan (O'Reilly).

Besides the JavaScript coverage, this appendix includes a section on the tools (IDEs, debuggers, web inspectors, and more) that will make your development process more productive.

JavaScript: A Brief History

The JavaScript programming language was designed in 1995 by Brendan Eich, who was working for Netscape Communications Corporation at the time. His goal was to allow developers to create more interactive web pages. Initially the name of this language was

Mocha and then LiveScript. Finally, Netscape agreed with Sun Microsystems, the creator of Java, to rename it to JavaScript.

A year later, the language was given to the international standards body Ecma, which formalized the language into *ECMAScript standard* so that other web browser vendors could create their implementation of this standard. JavaScript is not the only language that was created based on the ECMAScript specification; ActionScript is a good example of another popular dialect of ECMAScript.

To learn more about the history of JavaScript from the source, watch Brendan Eich's presentation "JavaScript at 17" at O'Reilly's conference Fluent 2012.

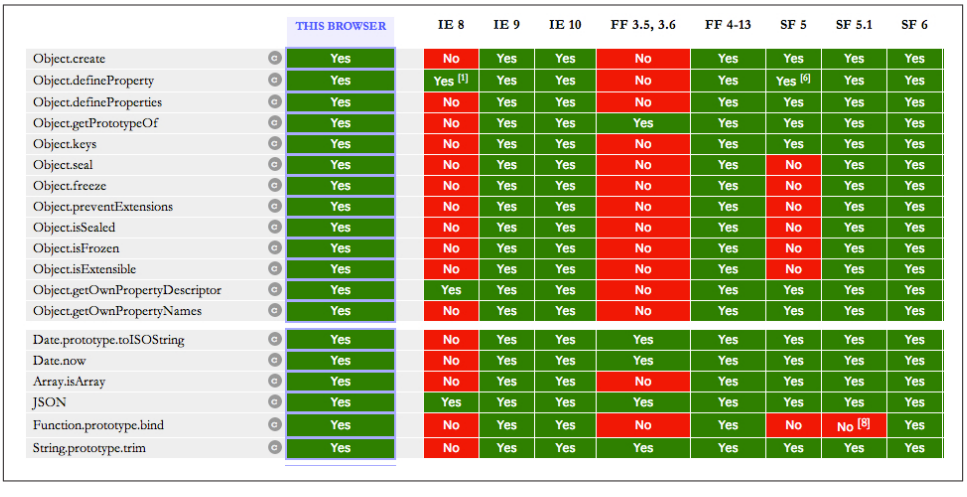
The vast majority of today's JavaScript code is being executed by web browsers, but there are JavaScript engines that can execute JavaScript code independently. For example, Google's V8 JavaScript engine implements ECMAScript 5 and is used not only in the Chrome browser, but can run in a standalone mode and can be embedded in any C++ application. Using the same programming language on the client and the server is the main selling point of Node.js, which runs on top of V8. Oracle's Java Development Kit (JDK) 8 will include the JavaScript engine Nashorn that not only can run on both the server and client computers, but also gives you the capability to embed the fragments of JavaScript into Java programs.

In the '90s, JavaScript was considered a second-class language, used mainly for making web pages more attractive. In 2005, the techniques known as Ajax (see ???) made a significant impact in the way web pages were built. With Ajax, the specific content inside a web page could be updated without having to make a full page refresh. For example, Google's Gmail inserts just one line at the top of your input box whenever a new email arrives—it doesn't usually re-retrieve the entire content of your inbox from the server and definitely doesn't re-render the web page.

Ajax gave a second birth to JavaScript. But the vendors of web browsers were not eager to implement the latest specifications of ECMAScript. Browsers' incompatibility and lack of good development tools prevented JavaScript from becoming the language of choice for web applications. Let's not forget about the ubiquitous Flash Player—a popular VM supported by all desktop web browsers. Rich Internet applications written in ActionScript were compiled into the byte code and executed by Flash Player on the user's machine, inside the web browser.

If Ajax saved JavaScript, rapid proliferation of tablets and smartphones made it really hot. Today's mobile devices come equipped with modern web browsers, and in the mobile world, there is no need to make sure that your web application will work in the four-year-old Internet Explorer 8. Adobe's decision to stop supporting Flash Player in mobile web browsers is yet another reason to turn to JavaScript if your web application has to be accessed from smartphones or tablets.

ECMAScript, 5th Edition, was published in 2009 and is currently supported by all modern web browsers. If you are interested in discovering whether specific features of ECMAScript 5 are supported by a particular web browser, check the latest version of the ECMAScript 5 compatibility table. At the time of this writing, the snapshot of the Chrome Browser v22 looks like Figure A-1.



	THIS BROWSER	IE 8	IE 9	IE 10	FF 3.5, 3.6	FF 4-13	SF 5	SF 5.1	SF 6
Object.create	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes
Object.defineProperty	Yes	Yes [1]	Yes	Yes	No	Yes	Yes [6]	Yes	Yes
Object.defineProperties	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes
Object.getPrototypeOf	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Object.keys	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes
Object.seal	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
Object.freeze	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
Object.preventExtensions	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
Object.isSealed	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
Object.isFrozen	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
Object.isExtensible	Yes	No	Yes	Yes	No	Yes	No	Yes	Yes
Object.getOwnPropertyDescriptor	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Object.getOwnPropertyNames	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes
Date.prototype.toISOString	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Date.now	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Array.isArray	Yes	No	Yes	Yes	No	Yes	Yes	Yes	Yes
JSON	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Function.prototype.bind	Yes	No	Yes	Yes	No	Yes	No	No [8]	Yes
String.prototype.trim	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Figure A-1. ECMAScript 5 compatibility sample chart

JavaScript became the lowest common denominator available on thousands of different devices. Yes, JavaScript engines are not exactly the same on the thousands of devices that people use to log in to Facebook, but they are pretty close, and using some of the JavaScript frameworks spares you from worrying about their incompatibilities.

JavaScript is an interpreted language that arrives on the web browser as text. The JavaScript engine optimizes and compiles the code before its execution. The JavaScript engine is a part of the web browser, which loads and executes the JavaScript code embedded or referenced between the HTML tags `<script>` and `</script>`. JavaScript was originally created for web browsers, which were supposed to display whatever content successfully arrived. What if an image doesn't arrive from the server? You'll see a broken image icon. What if erroneous JavaScript code with syntax errors arrives at the browser? Well, the engine will try to execute whatever code arrives. End users might appreciate the browser's forgiveness when at least some content is available, but software developers should be ready to spend more time debugging (in multiple browsers) the errors that could have been caught by compilers in other programming languages.

JavaScript Variables

JavaScript is a weakly typed language, so developers don't have the luxury of a strong compiler's help that Java or C# developers enjoy. For those unfamiliar with weakly typed languages, let us explain. Imagine that if in Java or C#, instead of declaring variables of specific data types, everything was of type `Object`, and you could assign to it any value —a string, a number, or a custom object `Person`. This would substantially complicate the ability of the compiler to weed out all possible errors. You don't need to declare variables in JavaScript; just assign a value, and the JavaScript engine will figure out the proper type during the execution of your code. For example, the variable named `girlfriend` will have a data type of `String`:

```
girlfriendName="Mary";
```

Because we haven't used the keyword `var` in front of `girlfriend`, this variable has global scope, which is a big no-no. Creating your global variables in a web application that often includes multiple libraries from different vendors can easily create situations in which the variable's value is accidentally replaced by someone else's code. Besides, if the application uses concurrent execution (read about web workers in ???), using a global variable can lead to race conditions. Soon, you'll see how to limit the scope of an application's variables to avoid polluting global space.

Variables declared with the `var` keyword inside a function have local scope and are not visible from outside that function. Consider the following function declaration:

```
function addPersonalInfo(){
   var address ="123 Main Street";      // local String variable
   age=25;                              // global Number variable
   var isMarried = true;                // local boolean variable
   isMarried = "don't remember";        // now it's of String type
}
```

The variables `address` and `isMarried` are visible only within the function `addPersonalInfo()`. The variable `age` becomes global because the keyword `var` was omitted.

The variable `isMarried` changes its type from `Boolean` to `String` during the execution of the preceding script, and the JavaScript engine won't complain, assuming that the programmer knows what she's doing. So be ready for the runtime surprises and allocate a lot more time for testing than with programs written in compiled languages.

Adding JavaScript to HTML

Software developers either directly include the JavaScript code in the HTML document by placing it between the tags `<script>` and `</script>` or include a reference to the external location of the code (for example, a local filename or a URL) in the `src` attribute of the `<script>` tag. We usually place the `<script>` tags at the end of HTML file. The

reason is simple: your JavaScript code might be manipulating HTML elements, and you want them to exist by the time the script runs. The other way to ensure that the code will run only after the web page has loaded is by catching a window's `load` event. (You'll see an example later in this appendix, in the section "DOM Events" on page 60.) Some JavaScript frameworks might have their own approach to dealing with HTML content, and in ???, the main HTML file of the web application written with the Ext JS framework has `<script>` tags followed by empty `<body>` tags. But let's keep things simple for now.

The code samples for this book are available on GitHub. The authors of this book use the WebStorm IDE from JetBrains (see ??? for details).

Create a new project *appendix_a* in the WebStorm IDE. Then create a new file called *new_file.html* and add the following fragment between the `</body>` and `</body>` tags:

```
<h1>Hello World</h1>

<script>
   alert("Hello from JavaScript");
</script>
```

In WebStorm, right-click *new_file.html* in WebStorm, select Open in Browser, and you'll see the output shown in Figure A-2 in your web browser.

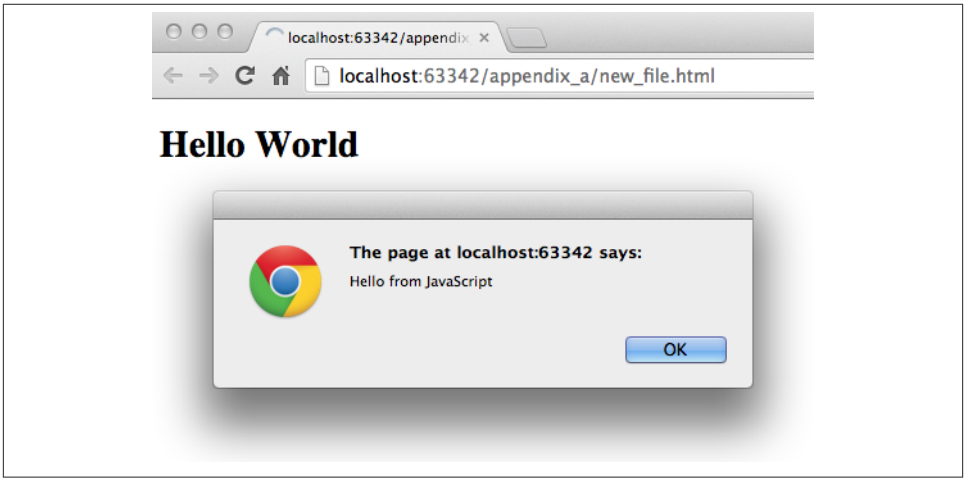


Figure A-2. MyFirstProject with appendix_a

Note that the `Alert` pop-up box is shown on top of the web page that already rendered its HTML component `<h1>`. Now move the preceding code from the `<body>` up to the

end of the <head> section and reopen *new_file.html*. This time, the picture is different; the alert box is shown before the HTML rendering is complete (see Figure A-3).

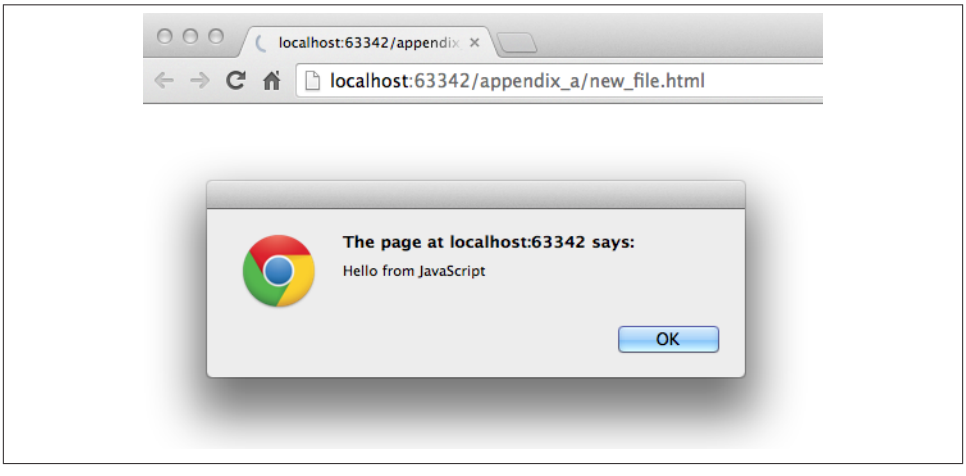


Figure A-3. Running HTML with JavaScript in the <head> section

This code sample doesn't cause any malfunctioning of the code, but if our JavaScript needs to manipulate the HTML elements, we'd run into issues of accessing nonexistent components. Beside this simple `Alert` box, JavaScript has `Confirm` and `Prompt` boxes, which give you the means to ask OK/Cancel types of questions or to request input from the user.

Debugging JavaScript in Web Browsers

The best way to learn any program is to run it step by step through a debugger. Some people appreciate using debuggers offered by the IDE, but we prefer to debug using great tools offered by the major web browsers:

- Firefox: Firebug add-on
- Chrome: Developer Tools
- Internet Explorer: F12 Developer Tools
- Safari: the menu Develop
- Opera: Dragonfly

We'll be doing most of the debugging either in Firebug or Chrome Developer Tools. Both provide valuable information about your code and are easy to use. To get Firebug, go to www.getfirebug.com and click the red Install Firebug button and then follow the instructions. In Firefox, open the Firebug panel from the View menu (see Figure A-4).

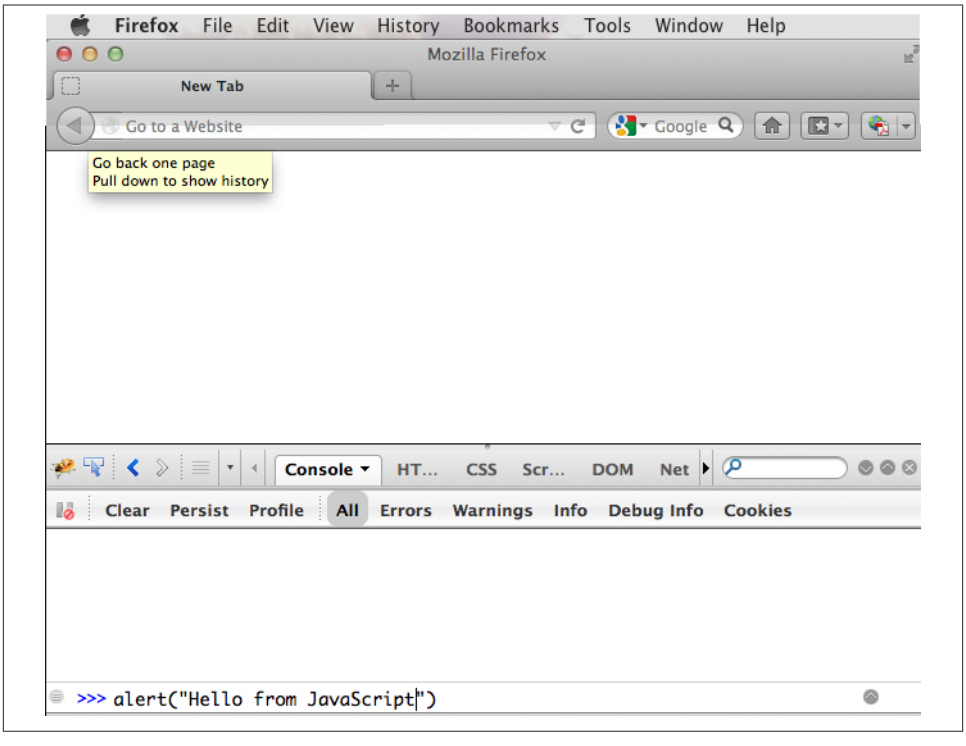


Figure A-4. The FireBug console

On the Firebug toolbar, select the Console option, enable the console, and then enter **alert("Hello from JavaScript")** after the >>> sign. You'll see the Alert box appear. To enter multiline JavaScript code, in the lower-right corner, click the little circle with a caret; Firebug will open a panel on the right, in which you can enter and run your JavaScript code.

This was probably the last example for which we used the `Alert()` pop-up box for debugging purposes. All JavaScript debuggers support `console.log()` for printing debug information. Consider the following example that illustrates the strict equality operator `===`. Yes, it's three equal signs in a row. This operator evaluates to true if the values are equal and the data types are the same:

```
var age=25;

var ageStr="25";

if (age==ageStr){
  console.log("The values of age and ageStr are equal");
}
```

```
if (age===ageStr){
 console.log("The values of age and ageStr are strictly equal");
} else{
 console.log ("The values of age and ageStr are not strictly equal");
}
```

Running this code in the Firebug console produces the output shown in Figure A-5.

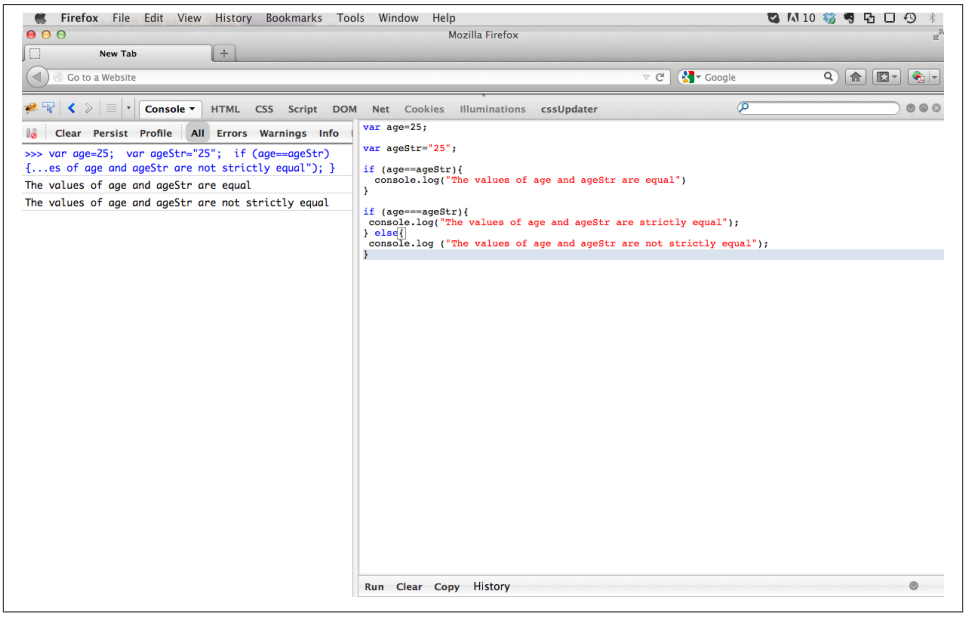


Figure A-5. Using console.log() for the debug output

You can also use `console.info()`, `console.debug()`, and `console.error()`; thus, the debuggers might highlight the output with different colors or mark it with different icons.

For more information about debugging JavaScript, refer to the code samples illustrated in Figures A-6 and A-7.

JavaScript Functions: A Gentle Introduction

JavaScript can be called an *object-oriented language* because an object can inherit existing functionality from another object, and you can encapsulate the data and restrict the data access. You can't do it as simply as in classical object-oriented languages, but it is possible. Now comes the chicken or the egg dilemma of what should be explained first: the syntax of functions or the creation of objects? Understanding objects is needed for some of the function code samples, and vice versa. We'll begin with simple function use cases and then switch to objects as needed.

Many readers have experience with object-oriented languages such as Java or C#, in which classes can include *methods* implementing required functionality. Then, these methods can be invoked with or without instantiation of the objects. If a JavaScript object includes functions, they are called *methods*. But JavaScript functions don't have to belong to an object. You can just declare a function and invoke it, like this:

```
//Function declaration
function calcTax (income, dependents){
   var tax;
   // Do stuff here
   return tax;
}

//Function invocation
calcTax(50000, 2);
var myTax = calcTax(50000,2);
```

The data types of the function parameters `income` and `dependents` are not specified. We can only guess that they are numbers based on their names. If a software developer doesn't bother giving meaningful names to function parameters, the code becomes difficult to read.

After the function `calcTax()` is invoked and complete, the variable `myTax` will have the value returned by the function.

Another important thing to notice is that our function has a name, `calcTax`. But this is not always the case. In JavaScript, functions can be *anonymous*. You'll see an example of anonymous functions in the function expressions that follow (note the absence of a name after the keyword `function`).

If you see a line of code in which the keyword `function` is preceded by any other character, this is not a function declaration, but a function expression.

Consider the following variation of the tax calculation sample:

```
//Function expression
var doTax=function (income, dependents){
        //do stuff here
   return tax;
}

//Function invocation
var myTax=doTax(50000,2);
```

In this code, the `function` keyword is used in the expression; we assign the anonymous function to the variable `doTax`. After this assignment, just the text of the function is assigned to the variable `doTax`—the anonymous function is not invoked just yet. It's important to understand that even though the code of this anonymous function ends with `return tax;`, actually, the tax calculation and return of its value does not happen until `doTax()` is invoked. Only then is the function evaluated, and the variable `myTax` will get whatever value this function returns.

Yet another example of a function expression is its placement inside the *grouping operator*—parentheses, as shown in the next code snippet. As in an arithmetic expressions, this means that the content inside the expression has to be evaluated first and then used in the expression:

```
(function calcTax (income, dependents){
   // Do stuff here
});
```

The outermost parentheses hide its internal code from the outside world, creating a scope or a closed ecosystem in which the function's code will operate. Try to add a line invoking this function, after the last line in the preceding code sample—for example, `calcTax(50000,2)`—and you'll get an error: `calcTax is not defined`. There is a way to expose some of the internal content of such a *closure*, and you'll see how to do it later in this appendix.

If you take away the outermost parentheses and the closing semicolon, you'll get a function declaration, which is subject to *hoisting* (we'll explain this soon). Function expressions are usually part of a larger expression. For example, if you add parentheses to the end of this expression, you'll get a *self-invoked* function. This extra pair of parentheses will cause the function expression located in the first set of parentheses to be executed right away:

```
(function calcTax (income, dependents){
   // Do stuff here
})();
```

The topic of function declaration versus function expressions is one of those fuzzy JavaScript areas that can cause unexpected behavior of your code. Angus Croll published a well-written article on this subject.

JavaScript Objects: A Gentle Introduction

JavaScript objects are simply unordered collections of properties. You can assign new properties or delete existing properties from objects during runtime whenever you please. In classical object-oriented languages, there are *classes* and there are *objects*. However, JavaScript doesn't have classes.

The ECMAScript 6 specification will include classes, too, but because it's a work in progress, we won't consider them as something useful in today's world. If you'd like to experiment with the upcoming features of JavaScript, download the Chrome Canary browser, go to `chrome:flags`, and then enable experimental JavaScript. Chrome Canary should be installed on the computer of any HTML5 developers. You can use today those features that will be officially released in Chrome Developer Tools in about three months.

In JavaScript, you can create objects by using one of the following methods:

- Using object literals
- Using `new Object()` notation
- Using `Object.create()`
- Using *constructor functions* and a `new` operator

Technically, other APIs can implicitly create objects—for example, `JSON.parse()`—but let's keep things simple.

In JavaScript everything is an object. Think of `Object` as of a root of the hierarchy of all objects used in your program. All your custom objects are descendants of `Object`.

Object Literals

The easiest way to create a JavaScript object is by using *object literal notation*. The following code sample begins with the creation of an empty object:

```
var t = {}            // create an instance of an empty object
```

The following line of code creates an object with one property, `salary`, and assigns the value 50000 to it:

```
var a = {salary: 50000}; // an instance with one property
```

Next, the instance of one more object is created, and the variable `person` points at it:

```
// Store the data about Julia Roberts
var person = { lastName: "Roberts",
               firstName: "Julia",
                     age: 42
             };
```

This object has three properties: `lastName`, `firstName`, and `age`. Note that in object literal notation, the values of these properties are specified with a colon. You can access the properties of this person by using dot notation—for example, `person.LastName`. But with JavaScript, there is yet another way for you to access object properties, by using square-bracket syntax—for example, `person["lastName"]`. In the next code sample, you'll see that using square brackets is the only way to access the property:

```
 var person = {
       "last name": "Roberts",
       firstName: "Julia",
             age: 42};

var herName=person.lastName;          // ❶

console.error("Hello " + herName);    // ❷

herName=person["last name"];           // ❸

person.salutation="Mrs. ";

console.log("Hello "+ person.salutation + person["last name"]); // ❹
```

❶ The object *person* doesn't have a property `lastName`, but no error is thrown.

❷ This prints "Hello undefined."

❸ An alternative way of referring to an object property.

❹ This prints "Hello Mrs. Roberts."

It's a good idea to keep handy a style guide of any programming language, and we know of two such documents for JavaScript. Google has published its version of a JavaScript style guide at *http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml*. A more detailed Airbnb JavaScript Style Guide is available as a GitHub project at *https://github.com/airbnb/javascript*. And the GitHub version of the JavaScript style guide is located at *https://github.com/styleguide/javascript*.

Nesting object literals

Objects can contain other objects. If a property of an object literal is also an object, you just need to specify the value of this property in an extra pair of curly braces. For example, you can represent the telephone of a person as an object having two properties: the type and the number. The following code snippet stores a work phone as a *nested object* inside the person's object. Run this code in the Firebug console, and it will print "Call Julia at work 212-555-1212."

```
var p = { lastName: "Roberts",
                firstName: "Julia",
                age: 42,
                phone:{
                      type: "work",
                      numb: "212-555-1212"
                 }
            };
console.log("Call " + p.firstName + " at " + p.phone.type + " " + p.phone.numb );
```

What if a person has more then one phone? We can change the name of the property `phone` to `phones` and instead store an array of objects. JavaScript arrays are surrounded by square brackets, and they are zero based. The following code snippet will print "Call Julia at home 718-211-8987."

```
var p = { lastName: "Roberts",
                firstName: "Julia",
                age: 42,
                phones:[{
                      type: "work",
                      numb: "212-555-1212"
                 },
                 {
                      type: "home",
                      numb: "799-211-8987"

                 }]
            };
console.log("Call " + p.firstName + " at " + p.phones[1].type + " "
                                           + p.phones[1].numb );
```

Defining methods in object literals

Functions defined inside objects are called *methods*. Defining methods in object literals is similar to defining properties: provide a method name followed by a colon and the function declaration. The code snippet that follows declares a method `makeAppointment()` to our object literal. Finally, the line `p.makeAppointment();` invokes this new method, which will print the message on the console that Steven wants to see Julia and will call at such-and-such number:

```
var p = { lastName: "Roberts",
                firstName: "Julia",
                age: 42,
                phones:[{
                      type: "work",
                      numb: "212-555-1212"
                 },
                 {
                      type: "home",
                      numb: "718-211-8987"

                 }],
                makeAppointment: function(){
                    console.log("Steven wants to see  " + this.firstName +
                                ". He'll call at " + this.phones[0].numb);
                }
            };

p.makeAppointment();
```

Because we already started using arrays, it's worth mentioning that arrays can store any objects. You don't have to declare the size of the array up front and can create new arrays as easily as `var myArray=[]` or `var myArray=new Array()`. You can even store function declarations as regular strings, but they will be evaluated on the array initialization. For example, during the `greetArray` initialization, the user will see a prompt asking her to enter her name, and, when it's done, the `greetArray` will store two strings. The output of the following code fragment looks like "Hello, Mary":

```
var greetArray=[
    "Hello",
    prompt("Enter your name", "Type your name here")
];

console.log(greetArray.join(","));
```

We've covered object literals enough so that you can begin using them. ??? covers JSON, a popular data format used as replacement for XML in the JavaScript world. You will

see that the syntax of JSON and JavaScript object literals are similar. Now we'll spend a little bit of time delving into JavaScript functions, and then back to objects again.

Constructor Functions

JavaScript functions are more than just named pieces of code that implement certain behavior. They also can become objects themselves by the magic of the `new` operator. To make things even more intriguing, the function calls can have memories, which is explained in the section "Closures" on page 43.

If a function is meant to be instantiated with the `new` operator, it's called a *constructor function*. If you are familiar with Java or C#, you understand the concept of a class constructor that is being executed only once during the instantiation of a class. Now imagine that there is only a constructor, without any class declaration that still can be instantiated with the `new` operator, as in the following example:

```
function Person(lname, fname, age){
         this.lastName=lname;
         this.firstName=fname;
         this.age=age;
};

// Creating 2 instances of Person
var p1 = new Person("Roberts","Julia", 42);

var p2 = new Person("Smith", "Steven", 34);
```

This code declares the function `Person`, and after that, with the help of the new operator, it creates two instances of the `Person` object referenced by the variables `p1` and `p2`, accordingly.

According to common naming conventions, the names of the constructor functions are capitalized.

The JavaScript language doesn't support classes, and a constructor function is the closest concept to class in languages such as Java or C#. ??? discusses the Ext JS framework that extends JavaScript and introduces constructs similar to classes and classical inheritance.

Adding methods and properties to functions

Objects can have methods and properties, right? On the other hand, functions are objects. Hence, functions can have methods and properties, too. If you declare a function `marryMe()` inside the constructor function `Person`, `marryMe()` becomes a method of `Person`. This is exactly what we'll do next. But this time, we'll create an HTML file that

includes the <script> section, referencing the JavaScript code sample located in a separate file.

If you want a hands-on example, create a new file in your Aptana project by choosing File→New→File and give it the name *marryme.js*. When prompted, accept the suggested default JavaScript template and then key in the following content into this file:

```
function Person(lname, fname, age){
        this.lastName=lname;
        this.firstName=fname;
        this.age=age;

        this.marryMe=function(person){
              console.log("Will you marry me, " + person.firstName);
        };

};

var p1= new Person("Smith", "Steven");
var p2= new Person("Roberts", "Julia");

p1.marryMe(p2);
```

This code uses the keyword `this`, which refers to the object where the code will execute. If you are familiar with the meaning of `this` in Java or C#, it's similar, but not exactly the same, and we'll illustrate it in the section "Scope, or Who's This?" on page 34. The method `marryMe()` of one `Person` object takes an instance of another `Person` object and makes an interesting proposition: "Will you marry me, Julia?"

This time we won't run this code in the Firebug console but rather will include it in the HTML file. In WebStorm, create a new HTML file, *marryme.html*. Modify it to include the JavaScript file *marryme.js*, as shown here:

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8" />
        </head>

        <body>
                <h1>Making Proposal</h1>

                <script src="marryme.js"></script>
        </body>
</html>
```

Debugging JavaScript in Firebug

Right-click the file *marryme.html* and choose Open in Browser. In Firefox, you'll see a new web page open called Making Proposals. Open Firebug by using the View menu,

refresh the page, and then switch to the Firebug Script tab. A split panel appears; the JavaScript code from *marryme.js* is on the left, as shown in Figure A-6.

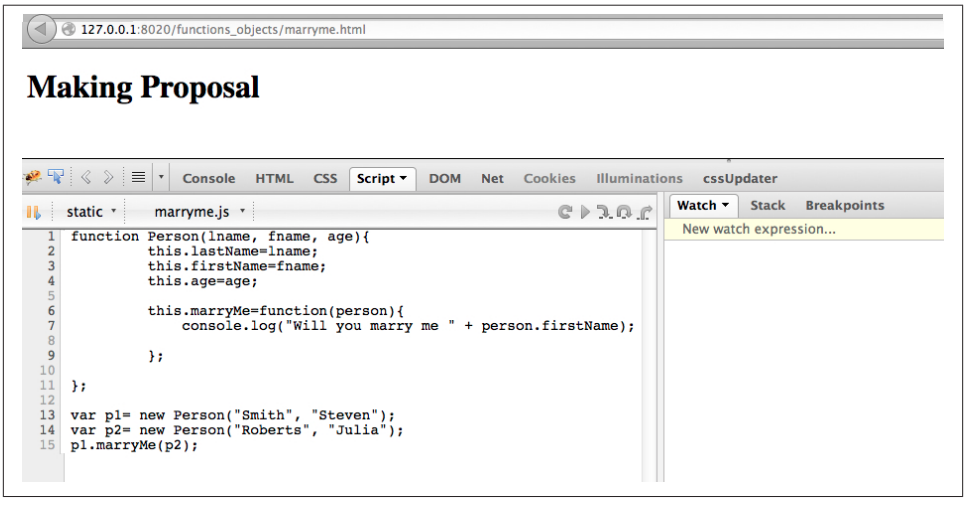


Figure A-6. Firebug's Script panel

Let's set a breakpoint inside the method `marryMe()` by clicking in the gray area to the left of line 7. You'll see a red circle that will reveal a yellow triangle as soon as your code execution hits this line. Refresh the content of the browser to rerun the script with a breakpoint. This time, the execution stops at line 7, and the right panel contains the runtime information about the objects and variables used by your program (see Figure A-7).

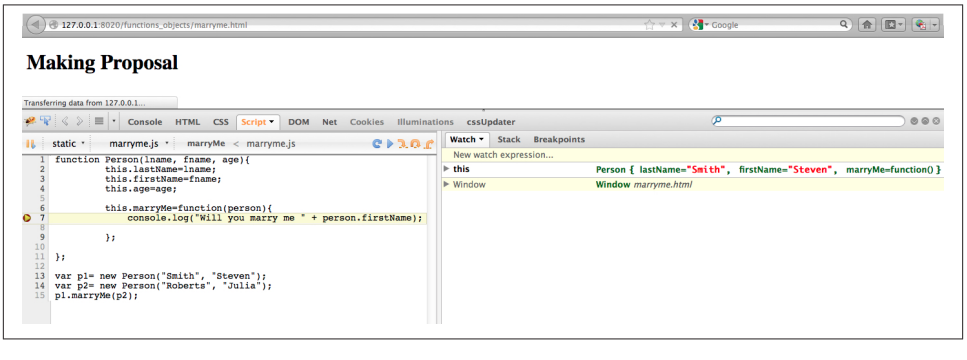


Figure A-7. Firebug's Script panel at a breakpoint

At the top of the left panel, you'll see standard debugger buttons with curved arrows (Step Into, Step Over, Step Out) as well as a triangular button to continue code execution.

The right panel depicts the information related to `this` and global `Window` objects. In Figure A-7, `this` represents the instance of the `Person` object represented by the variable `p1` (Steven Smith). To see the content of the object received by the method `marryMe()` you can add a watch variable by clicking the text "New watch expression…" and entering `person`—the name of the parameter of `marryMe()`. Figure A-8 shows the watch variable `person` (Julia Roberts) that was used during the invocation of the method `marryMe()`.

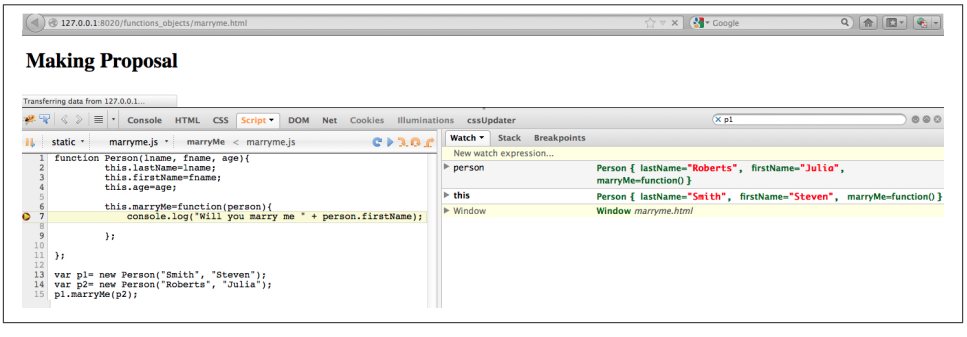


Figure A-8. Watching the person variable in the debugger

Now click Firebug's Net panel, which shows what goes over the network during communication between the web browser and web server. Figure A-9 shows a screenshot of the Net panel, in which we clicked the Headers tab for *marryme.html* and the Response tab of *marryme.js*. The code 200 for both files means that they arrived successfully to the browser. It also shows the IP address of the web server they came from, their sizes, and plenty of other useful information. Both the Script and Net panels of Firebug, or any other developers tools, are the best friends of any web developer.

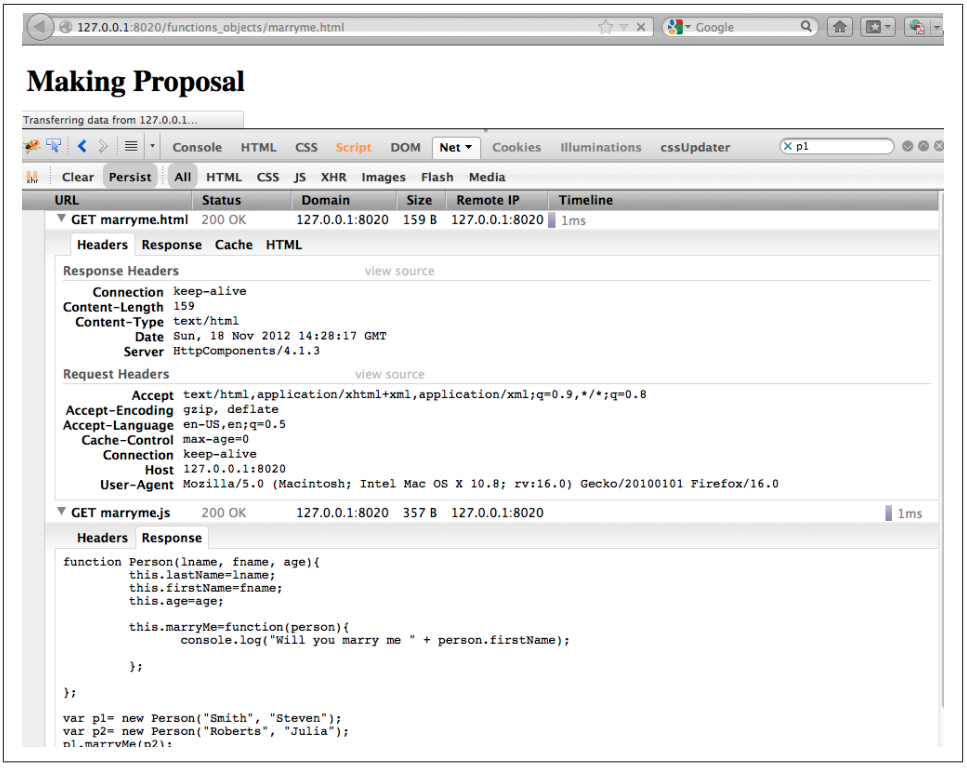


Figure A-9. Firebug's Net panel

We like Firebug, but testing and debugging should be done in several web browsers. Besides Firebug, we'll be using the excellent Google Chrome Developer Tools. Its menus and panels are similar, and we won't be including minitutorials on using such tools; you can easily learn them on your own.

You can find a tutorial on using Google Chrome Developer Tools at *https://developers.google.com/chrome-developer-tools/*. The cheatsheet for Chrome Developer Tools is located at *http://anti-code.com/devtools-cheatsheet/*. Finally, Google offers an online video course titled "Explore and Master Chrome DevTools."

Notes on Arrays

A JavaScript array is a grab bag of any objects. You don't have to specify in advance the number of elements to store, and there is more than one way to create and initialize array instances. The following code samples are self-explanatory:

```
var myArray=[];
    myArray[0]="Mary";
    myArray[2]="John";

// prints undefined John
console.log(myArray[1] + " " + myArray[2]);

var states1 = ["NJ", "NY", "CT", "FL"];

var states = new Array(4);  // size is optional

states[0]="NJ";

states[1]="NY";

states[2]="CT";

states[3]="FL";

// remove one array element
delete states[1];


// prints undefined CT length=4
console.log(states[1] + " " + states[2] + " Array length=" + states.length);

// remove one element starting from index 2
states.splice(2,1);

// prints undefined  FL length=3
console.log(states[1] + " " + states[2] + " Array length=" + states.length);
```

Removing elements with `delete` creates gaps in the arrays, whereas by using the array's method `splice()`, you can remove or replace the specified range of elements, closing gaps.

The next code sample illustrates an interesting use case, wherein we assign a string and a function text as array elements to `mixedArray`. During array initialization, the function `promt()` is invoked, the user is prompted to enter a name, and after that, two strings are stored in `mixedArray`—for example, "Hello" and "Mary."

```
var mixedArray=[
    "Hello",
    prompt("Enter your name", "Type your name here")
];
```

Prototypal Inheritance

JavaScript doesn't support classes, at least not until ECMAScript 6 becomes a reality. But you can create objects that inherit the properties and methods of other objects. By

default, all JavaScript objects are inherited from `Object`. Each JavaScript construction function has a special property called `prototype`, which points at this object's ancestor. If you want to create an inheritance chain whereby instances of the constructor function `ObjectB` *extends* `ObjectA` (similar to classical object-oriented languages), write one line of code such as `ObjectB.prototype=ObjectA;` (see Figure A-10).

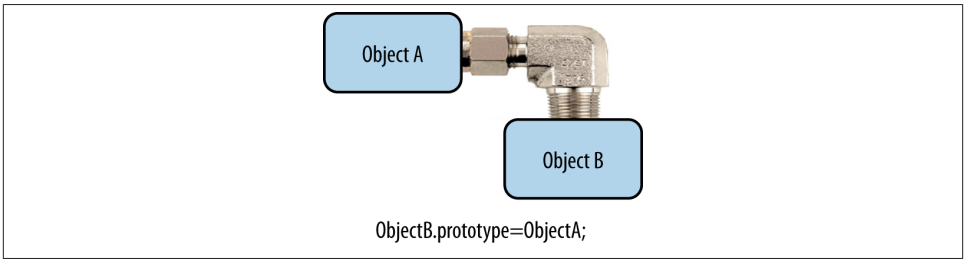


Figure A-10. Prototypal inheritance

Consider two constructor functions, `Employee` and `Person`, shown in the code snippet that follows. They represent two unrelated objects. But assigning the `Person` object to the `prototype` property of `Employee` creates an inheritance chain, and now the object `emp` will have all properties defined in both `Employee` and `Person`:

```
function Person(name, title){
        this.name=name;
        this.title=title;
        this.subordinates=[];
}

function Employee(name, title){
        this.name=name;
        this.title=title;
}

// All instances of Employee will extend Person
Employee.prototype = new Person();           // ❶

var emp = new Employee("Mary", "Specialist"); // ❷

console.log(emp);      // ❸
```

❶ Assign an ancestor of type person.

❷ Instantiate `Employee`.

❸ Print the object referred by `emp` to output `[object Object]`. It happens because each object has a method `toString()`, and if you want it to output useful information, override it. You'll see how to do this later in this section.

The preceding code results in code duplication, because the object referenced by the variable `emp` will have a pair of `name` and a pair of `title` properties. You'll see how to avoid such duplication a bit later, in the section "Avoiding Declaration Redundancy" on page 27.

The property `prototype` exists on constructor functions. After creating specific instances of such objects, you might see that these instances have another property called `proto`. At the time of this writing, this property is not a standard yet and won't be supported in some older browsers, but ECMAScript 6 will make it official. To illustrate the difference between `prototype` and `proto`, let's add the following piece of code to the previous code sample:

```
//Create an instance of Person and add property dependents
var p=new Person();
p.dependents=1;                 // ❶


var emp2=new Employee("Joe", "Father");

//This employee will have property dependents

emp2.__proto__=p;               // ❷

console.log("The number of Employee's dependents " + emp2.dependents);   // ❸
```

❶ Create an instance of `Person` and add an extra property, `dependents`, just for this instance.

❷ Assign this instance to the `__proto__` property of one instance.

❸ The code will properly print 1 as a number of dependents of the `Employee` instance represented by the variable `emp2`. The variable `emp` from the previous code snippet won't have the property `dependents`.

For a hands-on demonstration, open the file *WhoIsYourDaddy.html* (included in book code samples). Just for a change, this time we'll use Google Chrome Developer Tools by opening the browser's menu and choosing View→Developer→Developer Tools. Select the Sources tab and expand the panel on the left to select the file *WhoIsYourDaddy.js*. Set the breakpoint at the last line of the JavaScript, refresh the web page content, and add the watch expressions (click the + sign at the upper right) for the variables `p`, `emp`, and `emp2`. When the JavaScript code engine runs into `emp2.dependents`, it tries to find this property on the `Employee` object. If not found, the engine checks all the objects in the prototypal chain (in our case, it will find it in the object `p`) all the way up to the `Object`, if need be. Examine the values of the variable shown in Figure A-11.

If your program needs to work only with those properties that are defined on a specific object (not in its ancestors in the prototypal chain), use the method `hasOwnProperty()`.

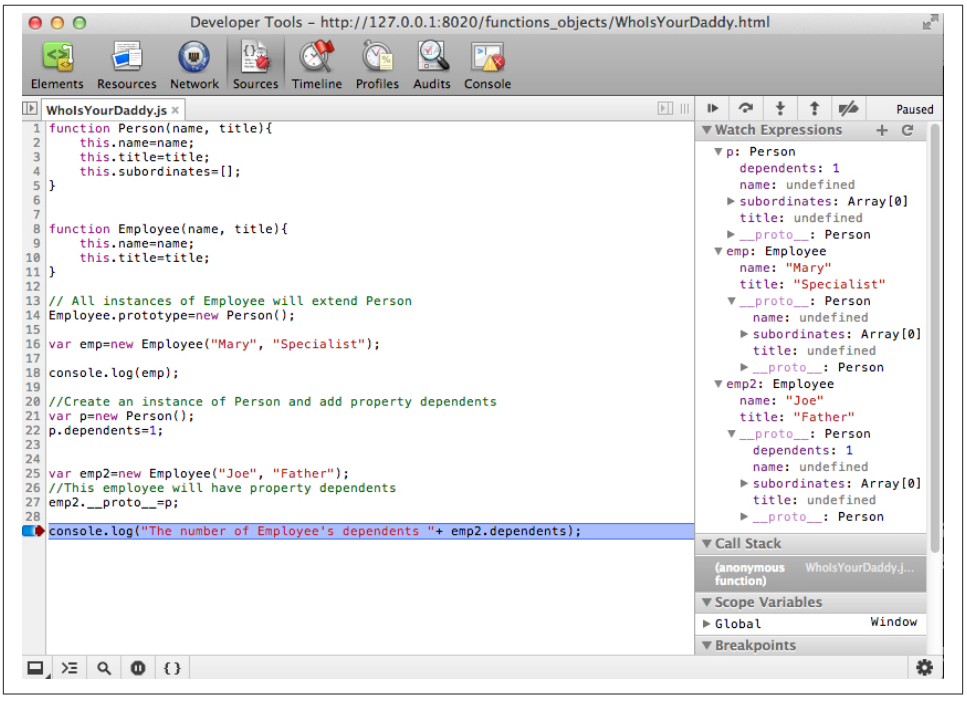


Figure A-11. The instance-specific proto variable

Note the difference in the content of the variables `__proto__` of the instances represented by `emp` and `emp2`. These two employees are inherited from two *different* objects `Person`. Isn't it scary? Not really.

Avoiding Declaration Redundancy

With prototypal inheritance, you can inherit one object from another, but it can lead to issues of redundancy and code duplication. If you take a closer look at the screenshot in Figure A-12, you'll see that the `Person` and `Employee` objects have redundant properties `name` and `title`. We'll deal with this redundancy in the section "Call and Apply" on page 37. But first, let's introduce and cure the redundancy in method declarations when prototypal inheritance is used.

Let's add a method `addSubordinate()` to the ancestor object `Person` that will populate its array `subordinates`. Who knows, maybe an object `Contractor` (descendant of a `Person`) will need to be introduced to the application in the future, so the ancestor's method `addSubordinate()` can be reused. *First, we'll do it the wrong way to illustrate the redundancy problem*, and then we'll do it right. Consider the following code:

```
// Constructor function Person
function Person(name, title){
        this.name=name;
        this.title=title;
        this.subordinates=[];

    // Declaring method inside the constructor function
        this.addSubordinate=function (person){
                this.subordinates.push(person)
        }
}

// Constructor function Employee
function Employee(name, title){
        this.name=name;
        this.title=title;
}

// Changing the inheritance of Employee
Employee.prototype = new Person();

var mgr =  new Person("Alex", "Director");
var emp1 = new Employee("Mary", "Specialist");
var emp2 = new Employee("Joe", "VP");

mgr.addSubordinate(emp1);
mgr.addSubordinate(emp2);
console.log("mgr.subordinates.length is " + mgr.subordinates.length);
```

The method `addSubordinate()` here is declared inside the constructor function `Person`, which becomes an ancestor of `Employee`. After instantiation of two `Employee` objects, the method `addSubordinate()` is duplicated for each instance.

Let's use the Google Chrome Developer Tools profiler to see the sizes of the objects allocated on the Heap memory. But first, we'll set up two breakpoints: one before, and one after creating our instances, as shown in Figure A-12.

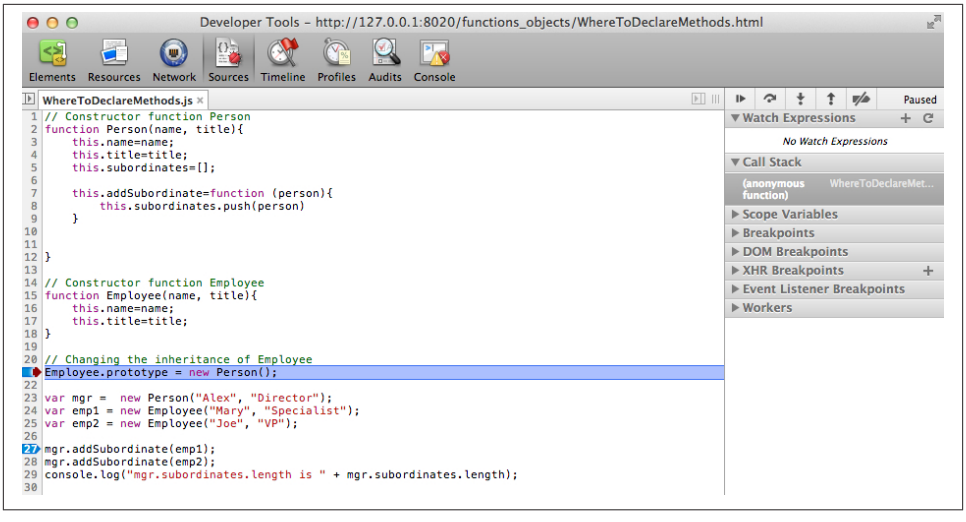


Figure A-12. Preparing breakpoints: take 1

When the execution of the code stops at the first breakpoint, we'll switch to the Profiler tab and take the first Heap snapshot. Upon reaching the second breakpoint, we'll take another Heap snapshot. Using the drop-down at the status bar, you can view the objects allocated between the snapshots 1 and 2. Figure A-13 depicts this view of the profiler. Note that the total size (the Shallow Size column) for the `Person` instances is 132 bytes. `Employee` instances *weigh* 104 bytes.

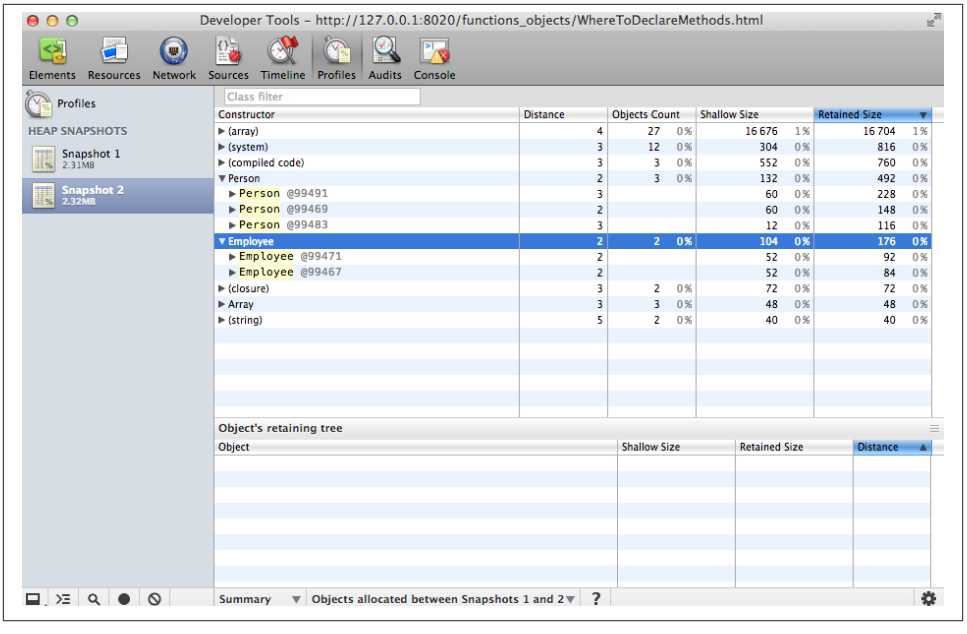


Figure A-13. Objects allocated between snapshots 1 and 2

Now we'll change the code to declare the method, not inside the `Person` constructor function, but on its prototype—and *this is the right way to declare methods in functions to avoid code duplication*:

```
// Constructor function Person
function Person(name, title){
        this.name=name;
        this.title=title;
        this.subordinates=[];

}

//Declaring method on the object prototype
Person.prototype.addSubordinate=function(subordinate){
   this.subordinates.push(subordinate);
   return subordinate;
}

// Constructor function Employee
function Employee(name, title){
        this.name=name;
        this.title=title;
}

// Changing the inheritance of Employee
Employee.prototype = new Person();
```

```
var mgr =  new Person("Alex", "Director");
var emp1 = new Employee("Mary", "Specialist");
var emp2 = new Employee("Joe", "VP");

mgr.addSubordinate(emp1);
mgr.addSubordinate(emp2);
console.log("mgr.subordinates.length is " + mgr.subordinates.length);
```

Similarly, we'll set up two breakpoints before and after object instantiation, as shown in Figure A-14.

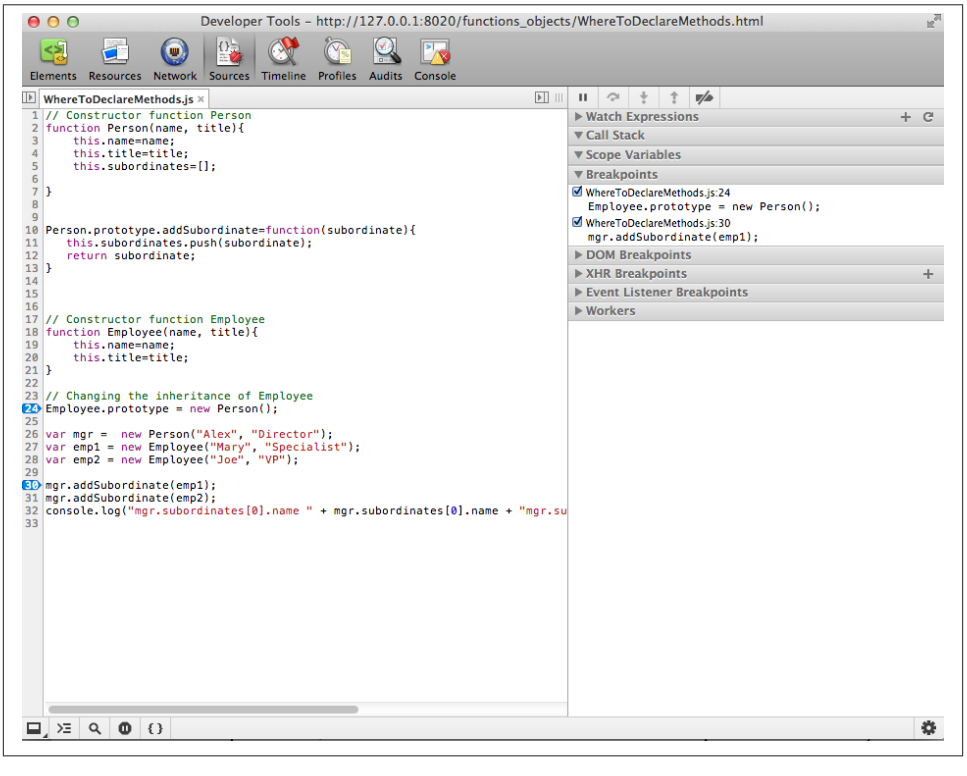


Figure A-14. Preparing breakpoints: take 2

Let's take two more profiler snapshots upon reaching each of the breakpoints. Although the size of the `Employee` instances remains the same (104 bytes), the `Person` instances become smaller: 112 bytes (see Figure A-15). Even though 20 bytes might not seem like a big deal, if you need to create hundreds or thousands of object instances, it adds up.

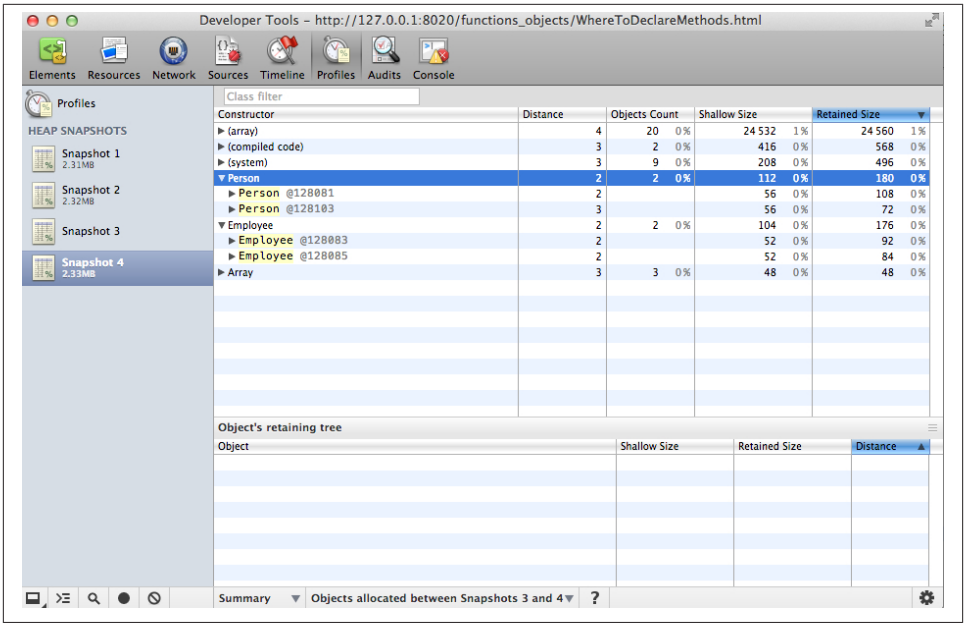


Figure A-15. Objects allocated between snapshots 3 and 4

So, if you need to declare a method on the object that will play a role of the ancestor, do it on the prototype level. The only exception to this rule occurs when the method needs to use an object-specific variable that's different for each instance; in that case, declare methods inside the constructors (for details, see the section "Closures" on page 43).

Implementing Missing Features with Polyfills

All modern web browsers support the function `Object.create()`, which creates a new object based on another prototype object and sets that new object's prototype to be the object passed in—for example, `var objectB=Object.create(objectA);`. What if you must support an older browser and need this "create by example" functionality? Of course, you can always create a custom, arbitrarily named function with similar functionality as the latest implementation of `Object.create()`. But the future-proof approach is to create the missing methods with the same signatures and on the same objects as the latest ECMAScript specification prescribes. In the case of `Object.create()`, you can use the implementation offered by Douglas Crockford:

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
```

```
    }
    newObject = Object.create(oldObject);
```

This approach of custom implementation of missing pieces according to the latest ECMAScript specifications or W3C drafts is known as *polyfills*. People who can't wait until browser vendors implement the newest functionality create their own cross-browser polyfills, and some of them submit their source code to the public domain. You can find a number of polyfills in the Git repository of the Modernizr project. The Can I use… website contains the current information about browser's support of the latest HTML5, JavaScript, and CSS features.

In ???, you can see how this framework offers its own class system that supports inheritance.

Method Overriding

Method overriding allows a subclass to replace (*override*) the functionality of a method defined in a superclass. Because JavaScript allows declaring methods on an object as well as on its prototype, overriding a method becomes really simple. The following code sample (see the file *overriding.js*) declares the method `addSubordinate()` on the prototype of the `Person` object, but then the object `p1` overrides this method:

```
function Person(name, title){

  this.name=name;
  this.title=title;
  this.subordinates=[];
}

Person.prototype.addSubordinate=function(person){

   this.subordinates.push(person);
   console.log("I'm in addSubordinate on prototype " + this);
}

var p1=new Person("Joe", "President");

    p1.addSubordinate=function(person){

    this.subordinates.push(person);
    console.log("I'm in addSubordinate in object " + this);
  }

    var p2 = new Person("Mary", "Manager")
```

```
p1.addSubordinate(p2);
```

Running this code prints only one line: "I'm in addSubordinate in object [object Object]." This proves that the method `addSubordinate()` on the prototype level is overridden. We can also improve this example by overriding the method `toString()` on the `Person` object. Just add the following fragment prior to instantiating `p1`:

```
Person.prototype.toString=function(){
   return "name:" + this.name +" title:" + this.title;
}
```

Now the code prints "I'm in addSubordinate in object name:Joe, title:President." Overriding the method `toString()` on objects is a common practice because it gives a textual representation of your objects.

Scope, or Who's This?

You are about to read one of the most confusing sections in this book. The confusion is caused by inconsistencies in JavaScript design and implementations by various browsers. Do you know what will happen if you remove the keyword `this` from the `toString()` method in the previous section? You'll get an error: the variable `title` is not defined. Without the keyword `this`, the JavaScript engine tries to find the variable `title` in the global namespace. Declaring and initializing the variable `title` outside the `Person` declaration eliminates this error, but this is not what we want to do. Misunderstanding the current scope can lead to errors that are difficult to debug.

Interestingly enough, replacing `this.name` with `name` doesn't generate an error but rather initializes the variable `name` with an empty string. Although `name` is not an officially reserved JavaScript keyword, there are articles in the blogosphere that don't recommend using the word `name` as a variable name. Keep this list of reserved words handy to avoid running into unpredictable behavior.

Let's consider several examples that illustrate the meaning of the `this` variable in JavaScript. The code sample that follows defines an object `myTaxObject` and calls its method `doTaxes()`. Notice two variables with the same name, `taxDeduction`—one of them has global scope and another belongs to `myTaxObject`. This little script is titled *ThisMafia.js*, and it was written for the mafia and will apply some under-the-table deductions for those who belong to Cosa Nostra:

```
var taxDeduction=300;      // global variable

var myTaxObject = {
```

```
    taxDeduction: 400,    // object property

    doTaxes: function() {
         this.taxDeduction += 100;

         var mafiaSpecial= function(){
           console.log( "Will deduct " + this.taxDeduction);
         }

         mafiaSpecial();  // invoking as a function
    }
}

myTaxObject.doTaxes();  //invoking method doTaxes
```

This code fragment illustrates the use of *nested functions*. The object method `doTaxes()` has a nested function `mafiaSpecial()`, which is not visible from outside `myTaxObject`, but it can certainly be invoked inside `doTaxes()`. What number do you think this code will print after the words "Will deduct"? Will it print three, four, or five hundred? Run this code in Firebug, Chrome Developer Tools, or any other way, and you'll see that it will print 300!

But this doesn't sound right, does it? The problem is that in JavaScript, the context where the function executes depends on the way it was invoked. In this case, the function `mafiaSpecial()` was invoked as a function (not a method) without specifying the object it should apply to, and JavaScript makes it operate in the global object; hence, the global variable `taxDeduction` having the value of 300 is used. So in the expression `this.taxDeduction` the variable `this` means global unless the code is operated in strict mode.

ECMAScript 5 introduced a restricted version of JavaScript called *strict mode*, which among other things places stricter requirements to variable declarations and scope identification. Adding `use strict` as the first statement of the method `doTax()` will make the context *undefined*, and it will print the error "this is undefined" and not 300. You can read about strict mode at Mozilla's developers site.

Let's make a slight change to this example to control what `this` represents. When the object `myTaxObject` was instantiated, its own `this` reference was created. The following code fragment stores the reference to `this` in an additional variable. `thisOfMyTaxObject` changes the game, and the expression `thisOfMyTaxObject.taxDeduction` evaluates to 500:

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,    // object property
```

```
    doTaxes: function() {
    var thisOfMyTaxObject=this;
         this.taxDeduction += 100;

         var mafiaSpecial= function(){
           console.log( "Will deduct " + thisOfMyTaxObject.taxDeduction);
         }

         mafiaSpecial();  // invoking as a function
    }
}

myTaxObject.doTaxes();  //invoking method doTaxes
```

You'll see a different way of running a function in the context of the specified object, using the special functions `call()` and `apply()`. But for now, consider one more attempt to invoke `mafiaSpecial()` shown in the following example that uses `this.mafiaSpecial()` notation:

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,    // object property

    doTaxes: function() {
         this.taxDeduction += 100;

         var mafiaSpecial= function(){
           console.log( "Will deduct " + this.taxDeduction);
         }

         this.mafiaSpecial();  // trying to apply object's scope
    }
}

myTaxObject.doTaxes();  //invoking method doTaxes
```

Run this code; you'll see the error "TypeError: this.mafiaSpecial is not a function," and rightly so. Take a closer look at the object `myTaxObject` represented by the variable `this`. The object `myTaxObject` has only two properties: `taxDeduction` and `doTaxes`. The function `mafiaSpecial` is hidden within the method `doTaxes` and can't be accessed via `this`.

After learning how to hide a function within an object, let's see how to do something quite the opposite: allowing an external method to run inside the context of an object.

Call and Apply

Visualize the International Space Station. Now, add to the picture an image of an approaching space shuttle. After attaching to the docking bay of the station, the shuttle's crew performs some functions on the station (a.k.a. the object) and then flies to another object or back to Earth. What does this have to do with JavaScript? It can serve as an analogy for creating a JavaScript function that can operate in the scope of any arbitrary object. For this purpose, JavaScript offers two special functions: `call()` and `apply()`. Both `call()` and `apply()` can invoke any function on any object. The only difference between them is that `apply()` passes parameters to a function as an array, whereas `call()` uses a comma-separated list.

Every function in JavaScript is an instance of the `Function` object. Both `call()` and `apply()` are defined in `Function`.

For example, you can invoke a function `calcStudentDeduction(income,numOfStudents)` in the context of a given object by using either `call()` or `apply()`. Note that with `call()`, you must list parameters explicitly, whereas with `apply()`, parameters are given as an array:

```
calcStudentDeduction.call(myTaxObject, 50000, 2);

calcStudentDeduction.apply(myTaxObject, [50000, 2]);
```

In this example, you can reference the instance of `myTaxObject` as `this` from within the function `calcStudentDeduction()`, even though this is a function and not a method. You can rewrite the last example from the previous section to invoke `mafiaSpecial()`. The following code will ensure that `mafiaSpecial()` has `this` pointing to `myTaxObject` and will print on the console "Will deduct 500."

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,

    doTaxes: function() {
         this.taxDeduction += 100;

         var mafiaSpecial = function(){
           console.log( "Will deduct " + this.taxDeduction);
         }

         mafiaSpecial.call(this);  // passing context to a function
```

```
    }
}

myTaxObject.doTaxes();
```

Callbacks

Can you program without using `call()` and `apply()`? Sure you can, but with JavaScript, you can easily create callbacks. The callback mechanism lets you pass the code of one function as a parameter to another function for execution in the latter function's context. This is a very useful feature of the language. Imagine an object with a method `processData()`. Depending on the business logic, you can pass to this method (as an argument) different functions that will do actual data processing: these are callbacks.

Another example of callbacks is event handlers. If a user clicks this button, here's the name of the handler function to call:

```
`myButton.addEventListener("click", myFunctionHandler);`
```

It's important to understand that *you don't immediately call* the function `myFunctionHandler` here; you are just registering it as the function argument. If the user clicks `myButton`, the code of the callback `myFunctionHandler` will be given to the object `myButton` and will be invoked in the context of the `myButton` object. The functions `call()` and `apply()` exist exactly for this purpose.

Let's consider an example. Suppose that you need to write a function that will take two arguments: an array containing preliminary tax data and a callback function, which will be applied to each element of this array. The following code sample (*Callback.js*) creates `myTaxObject` that has two properties: `taxDeduction` and `applyDeduction`. The latter is a method with two parameters:

```
var myTaxObject = {

    taxDeduction: 400, // state-specific  deduction

    // this function takes an array and callback as parameters
    applyDeduction: function(someArray, someCallBackFunction){

        for (var i = 0; i < someArray.length; i++){

            // Invoke the callback
           someCallBackFunction.call(this, someArray[i]);
        }

    }
}

// array
var preliminaryTaxes=[1000, 2000, 3000];
```

```
// tax handler function
var taxHandler=function(currentTax){
                    console.log("Hello from callback. Your final tax is " +
                    (currentTax - this.taxDeduction));
                }

// invoking applyDeduction passing an array and callback
myTaxObject.applyDeduction(preliminaryTaxes, taxHandler);
```

This code invokes `applyDeduction()`, passing it the array `preliminaryTaxes`, and the callback function `taxHandler` that takes the `currentTax` and subtracts `this.taxDeduction`. By the time this callback is applied to each element of the array, the value of `this` will be known and this code will print the following:

```
Hello from callback. Your final tax is 600
Hello from callback. Your final tax is 1600
Hello from callback. Your final tax is 2600
```

You might be wondering, why pass the function to another object if we could take an array, subtract 400 from each of its elements, and be done with it? The solution with callbacks gives you an ability to decide which function to call during runtime and call it only when a certain event happens. Using callbacks, you can do *asynchronous processing*. For example, you make an asynchronous request to a server and register the callback to be invoked if a result comes back. The code is not blocked and doesn't wait until the server response is ready. Here's an example from Ajax: `request.onreadystatechange=myHandler`. You register the `myHandler` callback but don't immediately call it. JavaScript functions are objects, so get used to the fact that you can pass them around as you'd be passing any objects.

Hoisting

A variable scope depends on where it was declared. You already had a chance to see that a variable declared inside a function with the keyword `var` is visible only within this function and any function declared within it. With some programming languages, you can narrow the scope even further. For example, in Java, declaring a variable inside any block of code surrounded with curly braces makes it visible only within that block. In JavaScript, it works differently. No matter where in the function you declare the variable, its declaration will be *hoisted* to the top of the function, and you can use this variable anywhere within the function.

Hoisting variables

The following code snippet will print 5, even though the variable `b` has been declared inside the `if` statement. Its declaration has been hoisted to the top:

```
function test () {
    var a=1;
```

```
    if(a>0) {
        var b = 5;
    }
    console.log(b);

}

test();
```

Let's make a slight change to this code to separate the variable declaration and initialization. The following code has two `console.log(b)` statements; the first one will output `undefined`, and the second will print 5, just as in the previous example:

```
function test () {
    var a=1;

    console.log(b);  // b is visible, but not initialized

    if(a>0) {
        var b;
    }

    b=5;

    console.log(b);  // b is visible and initialized
}

test();
```

Due to hoisting, JavaScript doesn't complain when the first `console.log(b)` is invoked. It knows about the variable `b`, but its value is `undefined` just yet. By the time the second `console.log(b)` is called, the variable `b` was initialized with the value of 5. Just remember that hoisting applies only to variable declarations and doesn't interfere with your code when it comes to initialization.

Hoisting functions

You can hoist JavaScript function declarations, too, as is illustrated in the following code sample:

```
function test () {
    var a=1;

    if(a>0) {
        var b;
    }

    b=5;

    printB();
```

```
        function printB(){
            console.log(b);
        }
    }

    test();
```

This code will print 5. We can call the function `printB()` here because its declaration was hoisted to the top. But the situation changes if instead of a function declaration we use a function expression. The following code will give you the error "PrintB is not a function":

```
    function test () {
        var a=1;

        if(a>0) {
            var b;
        }

        b=5;

        printB();

        var printB = function(){
            console.log(b);
        }

    }

    test();
```

Notice that the error doesn't complain about `printB` being undefined because the variable declaration was hoisted, but because the function expression wasn't, the JavaScript engine doesn't know yet that `printB` will become a function rather soon. Anyway, moving the invocation line `printB()` to the bottom of the function `test()` cures this issue.

Function expressions are not hoisted, but the variables to which they are assigned (if any) are hoisted.

Function Properties

Functions, like any other objects, can have properties. You can attach any properties to a `Function` object, and their values can be used by all instances of this object. Static variables in programming languages with classical inheritance is the closest analogy to function properties in JavaScript.

Let's consider an example of a constructor function `Tax`. Let's assume that we have an accounting program that can create multiple instances of `Tax`—one per person. Suppose that this program will be used in a Florida neighborhood with predominantly Spanish-speaking people. The following code (see the file *FunctionProperties.js*) illustrates the case in which the method `doTax()` can be called with or without parameters:

```
function Tax(income, dependents){
    this.income=income;              // instance variable
    this.dependents=dependents;      // instance variable

    this.doTax = function calcTax(state, language){
           if(!(state && language)){     // ❶
              console.log("Income: " + this.income + " Dependents: "+
                                                      this.dependents
              + " State: " + Tax.defaults.state + " language:" +
                                                    Tax.defaults.language);
           } else{                       // ❷
              console.log("Income: " + this.income + " Dependents: "+
                                                      this.dependents
              + " State: " + state + " language:" + language);
           }
    }
}

Tax.defaults={                           // ❸
     state:"FL",
     language:"Spanish"
};

// Creating 2 Tax objects
var t1 = new Tax(50000, 3);
    t1.doTax();                          // ❹
var t2 = new Tax(68000, 1);
    t2.doTax("NY","English");            // ❺
```

❶ No state or language were given to the method `doTax()`.

❷ The state and language were provided as `doTax()` parameters.

❸ Assigning the object with two properties as a `defaults` property on `Tax`. The property `default` is not instance specific, which makes it static.

❹ Invoking `doTax()` without parameters—use `defaults`.

❺ Invoking `doTax()` with parameters.

This program produces the following output:

```
Income: 50000 Dependents: 3 State: FL language:Spanish
Income: 68000 Dependents: 1 State: NY language:English
```

You can add as many properties to the constructor function as needed. For example, to count the number of instances of the `Tax` object, just add one more property: `Tax.counter=0;`. Now add to the `Tax` function something like `console.log(Tax.counter++);` and you'll see that the counter increments on each instance creation.

If multiple instances of a function object need to access certain HTML elements of the DOM, add references to these elements as function properties so objects can reuse them instead of traversing the DOM (it's slow) from each instance.

Closures

A *closure* is one of those terms that is easier explained by examples. Formal definitions are not very helpful to first-timers. Here's the definiton of a closure from Wikipedia:

> In programming languages, a closure (also lexical closure or function closure) is a function or reference to a function together with a referencing environment—a table storing a reference to each of the non-local variables (also called free variables or upvalues) of that function. A closure—unlike a plain function pointer—allows a function to access those non-local variables even when invoked outside its immediate lexical scope.

It's not a very helpful definition, is it? Let's try to give a better one. Imagine a function that contains a private variable and a nested function. Is it possible to invoke the nested function from outside the outer one? And if it's possible, what does this inner function know about its surroundings?

Larry Ullman gives the following definition in *Modern Java Script* (Peachpit Press): "Closure is a function call with memory." We can offer you our version of what a closure is: A closure is a function call with strings attached.

Why Do We Need Closures?

In classical object-oriented languages, you create an object with a certain state and behavior and can pass it to a method of another object for further processing. In JavaScript, you can even pass a function to an object's method for further processing. But what if a function also needs to remember the state (the values of external variables) of the context where the function was defined?

Think of a closure as a function that remembers state. It's just a special type of object that can be passed between objects and use certain variables that didn't seem to be defined in the function's code. But they existed in the context where the function was defined.

Closures by Example

Now it's time to explain these mysterious definitions, and we'll do it by example. Consider the following code (see *closure1.js*) that is yet another example of implementing the tax collection functionality:

```
(function (){                // this is an anonymous function expression

    var taxDeduction = 500;  // private context to remember

      //exposed closure
      this.doTaxes=function(income, customerName) {

        var yourTax;

        if (customerName !== "Tony Soprano"){
          yourTax = income*0.05 - taxDeduction;
        } else{
          yourTax = mafiaSpecial(income);
        }

         console.log( "   Dear " + customerName + ", your tax is "+ yourTax);
         return yourTax;
      }

      //private function
      function mafiaSpecial(income){
          return income*0.05 - taxDeduction*2;
      }

})();    // Self-invoked function

// The closure remembers its context with taxDeduction=500
doTaxes(100000, "John Smith");
doTaxes(100000, "Tony Soprano");

mafiaSpecial();        // throws an error - this function is private
```

First, a self-invoking function creates an anonymous instance of an object in the global scope. It contains a private variable `taxDeduction`, a public method `doTaxes()`, and a private method `mafiaSpecial()`. Just by virtue of declaring `doTaxes` on `this` object, this method becomes exposed to the current scope, which is global in this example.

After that, we call the method `doTaxes()` twice. Note that the function `doTaxes()` uses the variable `taxDeduction` that was never declared there. But when `doTaxes` was initially declared, the variable `taxDeduction` with a value of 500 was already there. So the internal function "remembers" the context (the neighborhood) where it was declared and can use it for its calculations.

The algorithm of tax calculations makes `doTaxes()` call the function `mafiaSpecial()` if the customer's name is Tony Soprano. The function `mafiaSpecial()` is not visible from outside, but for insiders like `doTaxes()`, it's available. Here's what the preceding code example prints on the console:

```
Dear John Smith, your tax is 4500
Dear Tony Soprano, your tax is 4000
Uncaught ReferenceError: mafiaSpecial is not defined
```

Figure A-16 shows a screenshot taken when `doTaxes()` hit the breakpoint inside `doTaxes`. Note the right panel that shows what's visible in the Closure scope.

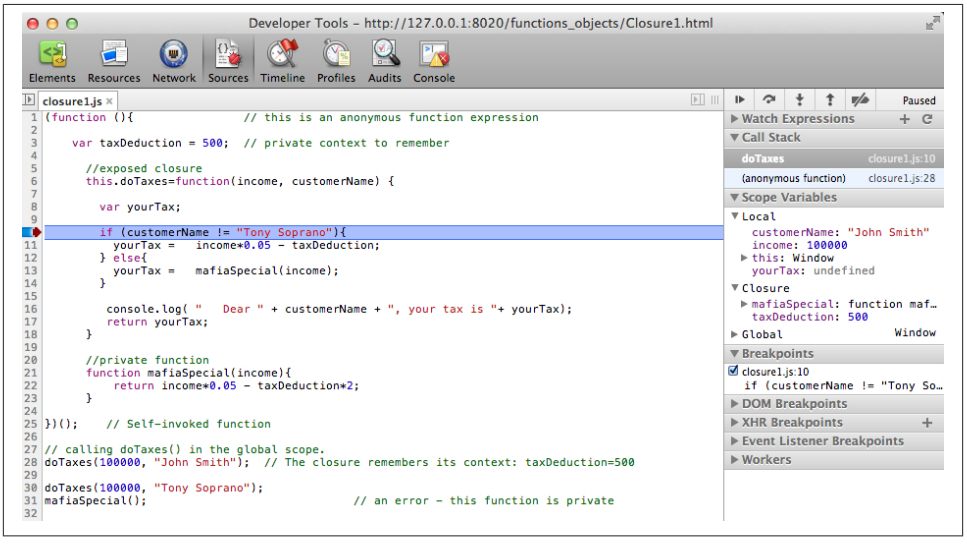


Figure A-16. Closure view in Chrome Developer Tools

JavaScript doesn't give you an explicit way to mark a variable as private. By using closures, you can get the same level of data hiding that you get from private variables in other languages. In the preceding example, the variable `taxDeduction` is local for the object enclosed in the outermost parentheses and can't be accessed from outside. But `taxDeduction` can be visible from the object's functions `doTaxes` and `mafiaSpecial`.

Figure A-17 gives yet another visual representation of our code sample. The self-invoked anonymous function is shown as a cloud that exposes only one thing to the rest of the world: the closure `doTaxes`.

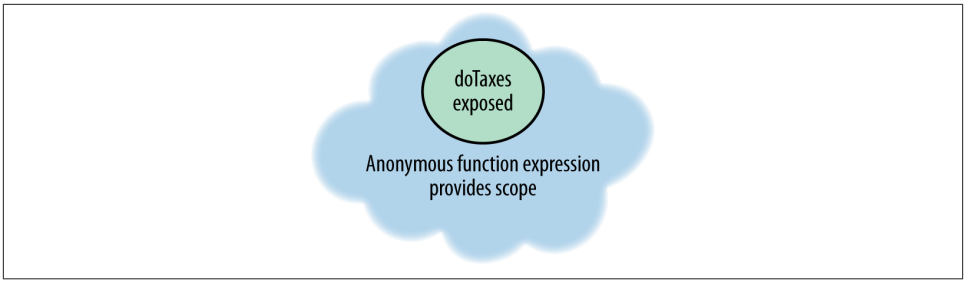


Figure A-17. The closure doTaxes

Let's consider a couple more cases of returning a closure to the outside world so it can be invoked later. Whereas the previous code sample exposes the closure by using `this.taxes` notation, the next two examples simply return the code of the closure by using the `return` statement. The following code (see *closure3.js*) declares a constructor function `Person`, adds a function `doTaxes()` to its prototype, and then finally creates two instances of the `Person` calling the method `doTaxes()` on each of them:

```
// Constructor function
function Person(name){

        this.name = name;

}

// Declaring a method that returns closure
Person.prototype.doTaxes= function(){

    var taxDeduction = 500;

      //private function
      function mafiaSpecial(income){
          return income*0.05 - taxDeduction*2;
      }

      //the code of this function is returned to the caller
      return function(income) {

        var yourTax;

        if (this.name !== "Tony Soprano"){
          yourTax =   income*0.05 - taxDeduction;
        } else{
          yourTax =   mafiaSpecial(income);
        }

         console.log( "My dear " + this.name + ", your tax is "+ yourTax);
         return yourTax;
      }
```

```
}();     // important parentheses!

//Using closure
var p1 = new Person("John Smith");
var result1 = p1.doTaxes(100000);

var p2 = new Person("Tony Soprano");
var result2 = p2.doTaxes(100000);
```

The calculated taxes in this example are the same as in the previous one: John Smith has to pay $4,500, whereas Tony Soprano has to pay only $4,000. But we used a different technique for exposing the closure. We want to make sure that you didn't overlook the parentheses at the very end of the function expression for `doTaxes`. These parentheses force the anonymous function to self-invoke; it will run into a `return` statement and will assign the code of the anonymous inner function that takes the parameter `income` to the property `doTaxes`. So when the line `var result1 = p1.doTaxes(100000);` calls the closure, the variable `result1` will have the value 4500. Remove these important parentheses, and the value of `result1` is not the tax amount, but the code of the closure itself—the invocation of the closure is not happening.

The following code fragment (see *closure2.js*) is yet another example of returning the closure that remembers its context:

```
function prepareTaxes(studentDeductionAmount) {

        return function (income) {          // ❶
           return income*0.05 - studentDeductionAmount;
        };

}

var doTaxes = prepareTaxes(300);         // ❷
var yourTaxIs = doTaxes(10000);          // ❸
console.log("Your tax is " + yourTaxIs);  // ❹
```

❶ When the function `prepareTaxes` is called, it immediately hits the `return` statement and returns the code of the closure to the caller.

❷ After this line is executed, the variable `doTaxes` has the code of the closure, which remembers that `studentDeductionAmount` is equal to 300.

❸ This is the actual invocation of the closure.

❹ The console output is "your tax is 200."

First, the closure is returned to the caller of `prepareTaxes()`, and when the closure is invoked, it remembers the values defined in its outer context. After looking at this code, you might say that nothing is declared in the closure's outside context! There is—by the time the closure is created, the value of `studentDeductionAmount` will be known.

Check the quality of your code with the help of JavaScript code-quality tools such as JSLint or JSHint.

Closures as Callbacks

Let's revisit the code from the previous section. That code shows how to pass an arbitrary function to another one and invoke it there by using `call()`. But if that version of the function `taxHandler` is not aware of the context in which it was created, the following version is. If in classical object-oriented languages you need to pass a method that knows about its context, you create an instance of an object that contains the method and the required object-level properties, and then you pass this wrapper-object to another object for processing. But because the closure remembers its context anyway, we can just pass a closure as an object. Compare the following code (see *callbackWithClosure.js*) with the code from "Callbacks" on page 38:

```
var myTaxObject = {

    // this function takes an array and callback as parameters
    applyDeduction: function(someArray, someCallBackFunction){

        for (var i = 0; i < someArray.length; i++){

            // Invoke the callback
           someCallBackFunction.call(this, someArray[i]);
        }

    }
}

// array
var preliminaryTaxes=[1000, 2000, 3000];


var taxHandler = function (taxDeduction){

// tax handler closure
        return function(currentTax){
                    console.log("Hello from callback. Your final tax is " +
                    (currentTax - taxDeduction));
                };
}

// invoking applyDeduction passing an array and callback-closure
myTaxObject.applyDeduction(preliminaryTaxes, taxHandler(200));
```

The last line of this example calls `taxHandler(200)`, which creates a closure that's being passed as a callback to the method `applyDeduction()`. Even though this closure is executed in the context of `myTaxObject`, it remembers that the tax deduction is 200.

Mixins

The need to extend capabilities of objects can be fulfilled by inheritance, but this is not the only way of adding behavior to objects. In this section, you'll see an example of something that would not be possible in object-oriented languages such as Java or C#, which don't support multiple inheritance. JavaScript makes it possible to take a piece of code and *mix it into any object* regardless of its inheritance chain. A *mixin* is a reusable code fragment that an object can borrow without the need to use inheritance. We'll illustrate this concept by example.

In the next code fragment, we define a function expression and assign it to a variable named `Tax`. This is a closure that includes the function `calcTax()` which knows the values of `income` and `state`. There is also an independent mixin, `TaxMixin`, with a couple of functions, `mafiaSpecial()` and `drugCartelSpecial()`. We want to blend this mixin into `Tax`. After this is done, the `Tax` object will have its original functionality—for example, `calcTax()`—as well as new "mafia and drug cartel" flavors. The following code is located in the file *mixins.js*:

```
// Defining a function expression
var Tax = function(income, state){
        this.income=income;
        this.state=state;

        this.calcTax=function(){
                var tax=income*0.05;
                console.log("Your calculated tax is " + tax)
                return tax;
        }
};


// Defining a mixin
var TaxMixin = function () {};

TaxMixin.prototype = {

  mafiaSpecial: function(originalTax){
    console.log("Mafia special:" + (originalTax - 1000));
  },

  drugCartelSpecial: function(originalTax){
     console.log("Drug Cartel special:" + (originalTax - 3000));
  }
```

```
};

// this function can blend TaxMixin into Tax
function blend( mainDish, spices ) {

  for ( var methodName in spices.prototype ) {
      mainDish.prototype[methodName] = spices.prototype[methodName];
  }
}

// Blend the spices with the main dish
blend( Tax, TaxMixin );

// Create an instant of Tax
var t = new Tax(50000, "NY");

var rawTax = t.calcTax();

// invoke a freshly blended method
t.mafiaSpecial(rawTax);
```

The function `blend()` loops through the code of the `TaxMixin` and copies all its properties into `Tax`. After the function `blend()` is finished, you can call on the `Tax` instance the newly acquired methods `mafiaSpecial()` and `drugCartelSpecial()`.

Mixins can be useful if you want to provide a specific feature to numerous objects without changing their inheritance chains. The other use case is if you want to prepare a bunch of small code fragments (think, spices) and add any combination of them to the various objects (dishes) as needed. Mixins give you a lot of flexibility in what you can achieve with minimum code, but they can decrease the readability of your code.

If you've read this far, you should have a good understanding of the syntax of the JavaScript language. Studying the code samples provided in this appendix has one extra benefit: now you can apply for a job as a tax accountant in a mafia near you.

JavaScript in the Web Browser

After learning all these facts and techniques about the language, you might be eager to see the real-world use of JavaScript. Slowly but surely, a web browser becomes the leading platform for development of the user interface. The vast majority of today's JavaScript programs primarily manipulate HTML elements of web pages. In this section, we'll be doing exactly this: applying JavaScript code to modify the content or style of HTML elements.

The Document Object Model (DOM)

DOM stands for *Document Object Model*. It's an object representing the hierarchy of HTML elements of a web page. Every element of the HTML document is loaded into

the DOM. Each DOM element has a reference to its children and siblings. When the DOM was invented, web pages were simple and static. The DOM was not meant to be an object actively accessed by code. This is the reason why on some heavily populated web pages, manipulating DOM elements can be slow. Most likely the DOM is the main target for anyone who's trying to optimize the performance of a web page.

If your web page is slow, analyze it by using YSlow, a tool built based on the Yahoo! rules for high-performance websites. Also, you can minimize and obfuscate your JavaScript code with the help of JavaScript Compressor.

When a web browser receives content, it performs the following activities:

- Adds arriving HTML elements to the DOM and lays out the content of the web pages
- Renders the UI
- Runs JavaScript that was included in the HTML
- Processes events

The amount of time spent on each activity varies depending the content of the page.

If you are interested in learning how browsers work, read the excellent write-up titled "How Browsers Work: Behind The Scenes of Modern Web Browsers."

Let's consider the operations that your application needs to be able to perform inside a web page:

- Programmatically find the required element by ID, type, or a CSS class
- Change styles of elements (show, hide, apply fonts and colors, and more)
- Process events that might happen to HTML elements (`click`, `mouseover`, and the like)
- Dynamically add or remove HTML elements from the page or change content
- Communicate with the server side (for example, submitting forms or making Ajax requests for data from the server)

Now you'll see some code samples illustrating the use of JavaScript for these operations. Even if you're using one of the popular JavaScript frameworks, your program will be

performing similar operations, applying the syntax prescribed by your framework of choice. So let's learn how it can be done.

Working with the DOM

If you want to change the appearance of an HTML page, you need to manipulate the DOM elements. Older web applications prepared HTML content on the server side. For example, a server-side Java servlet would compose and send to the client HTML whenever the application logic required a change to the appearance of the UI. The current trend is different: the client's code takes care of the UI rendering, and only the data goes back and forth between the client and the server. You can see how this works in more detail in ???, which explains the use of Ajax and JSON.

Earlier in this appendix, we talked about the global namespace where all JavaScript objects live unless they were declared with `var` inside the functions. If JavaScript code is running in a web browser, this global namespace is represented by a special variable `window`. It's an implicit variable, and you don't have to use it in your code, but whenever we say that a variable is global, we mean that it exists in the `window` object. For example, the following code prints "123 Main Street" twice:

```
var address ="123 Main Street";

console.log(address);
console.log(window.address);
```

The `window` object has many useful properties, including `cookie`, `location`, `parent`, and `document`. The variable `document` points at the root of the DOM hierarchy. Often your JavaScript code will find an element in the DOM first, and then it can read or modify its content. Figure A-18 is a screenshot from Firebug showing the fragment of a DOM of the simple web page *mixins.html*.

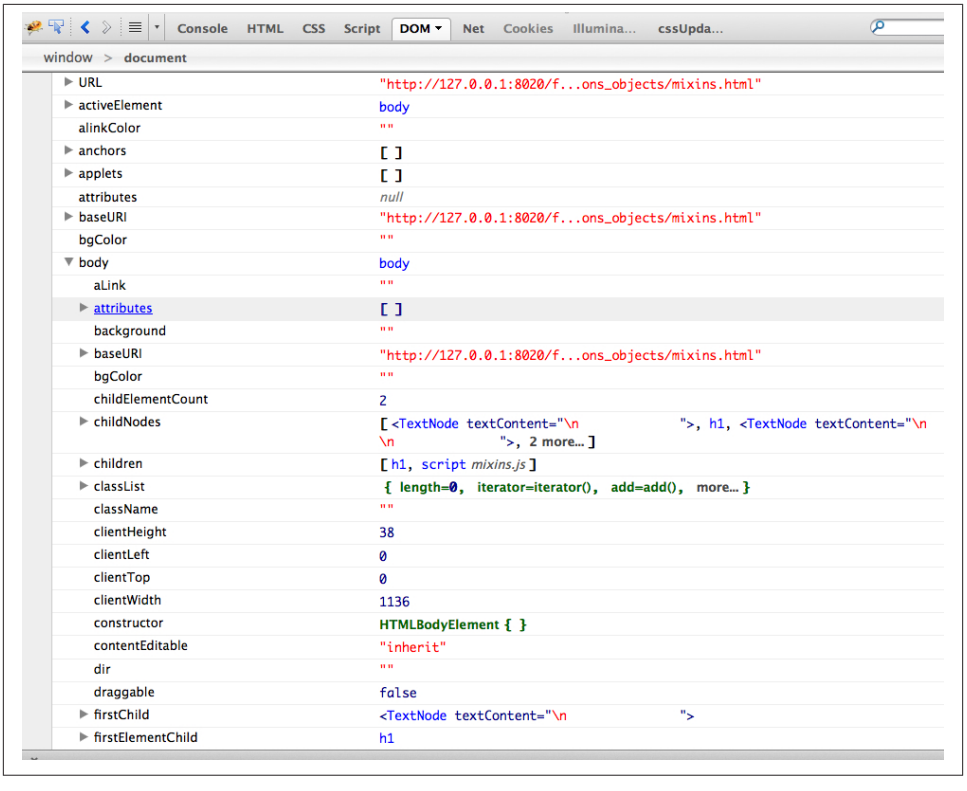


Figure A-18. Firebug's representation of the DOM

The following are some of the methods that exist on the `document` object:

`document.write(text)`
: Adds the specific text to the DOM. Careless use of the method `write()` can result in unpredictable results, if after changing the DOM the HTML content is still arriving.

`document.getElementById(id)`
: Gets a reference to the HTML element by its unique identifier.

`document.getElementsByTagName(tname)`
: Gets a reference to one or more elements by tag names; for example, a reference to all `<div>` elements.

`document.getElementsByName(name)`
: Gets a reference to all elements that have the requested value in their name attribute.

`document.getElementsByClassName(className)`
: Gets a reference to all elements that use specified CSS class(es), such as `document.getElementsByClassName('red text-left')`.

`document.querySelector(cssSelector)`
: Finds the first element that matches the provided CSS selector string. This comes in handy if you want to specify more-complex queries than just a class name; for example, `document.querySelector("style[type='text-left']");`.

`document.querySelectorAll(cssSelector)`
: Finds all elements that match the provided CSS selector string.

The next code sample contains the HTML `<span>` element that has an ID of `emp`. Initially, it contains an ellipsis, but when the user enters the name in the input text field, the JavaScript code finds the reference to this `<span>` element and replaces the ellipsis with the content of the input text field:

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8" />
        </head>

        <body>
        <h2>Selecting DOM elements</h2>

        <p>
                The employee of the month is <span id="emp">...</span>
        <br>
        <input type="button" value="Change the span value"
               onclick="setEmployeeOfTheMonth()"/>
        Enter your name  <input type="text" id="theName" />
        </p>

        <script>
           function setEmployeeOfTheMonth(){

                   var mySpan = document.getElementById("emp");

                   var empName = document.getElementsByTagName("input")[1];

                   mySpan.firstChild.nodeValue = empName.value;

                }
        </script>

        </body>
</html>
```

Note the input field of type `button`, which includes the `onclick` property that corresponds to the `click` event. When the user clicks the button, the browser dispatches the `click` event and calls the JavaScript function `setEmployeeOfTheMonth()`. The latter queries the DOM and finds the reference to the `emp` by calling the method `getElementById()`. After that, the method `getElementByTagName()` is called, trying to find all the references to the HTML `<input>` elements. This method returns an array because there could be more than one element with the same tag name on a page, which explains the use of array notation. The first `<input>` element is a button, and the second is the text field in which we're interested. Remember that arrays in JavaScript have zero-based indexes. Figure A-19 shows the web page after the user enters the name *Mary* and clicks the button.

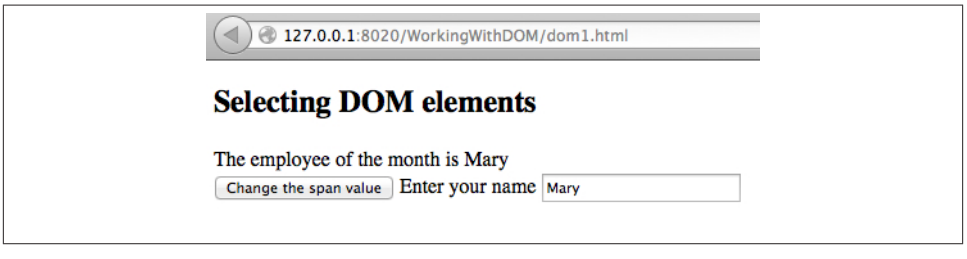


Figure A-19. Changing the content of the HTML element

While manipulating the content of your web page, you might need to traverse the DOM tree. The code example that follows shows you an HTML document which includes JavaScript that walks the DOM and prints the name of each node. If a node has children, the recursive function `walkTheDOM()` will visit each child:

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8" />
        </head>

    <body>
     <h1>WalkTheDom.html</h1>

     <p>
        Enter your name: <input type="text"
                                name="customerName" id="custName" />
     </p>

     <input type="button" value="Walk the DOM"
                          onclick="walkTheDOM(document.body, processNode)"/>

     <script>
                function walkTheDOM(node, processNode){
```

```
                processNode(node)
                 node = node.firstChild;

                        while(node){
                           // call wakTheDOM recursively for each child
                           walkTheDOM(node,processNode);
                           node = node.nextSibling;
                        }
            }

         function processNode(node){
            // the real code for node processing goes here

                console.log("The current node name is "+  node.nodeName);
         }
      </script>
   </body>
</html>
```

Our function `processNode()` just prints the name of the current node, but you could implement any code that your web application requires. Run this code in different browsers and check the output on the JavaScript console. Figure A-20 depicts two screenshots taken in the F12 Developer Tools in Internet Explorer (left) and Firebug running in Firefox (right).

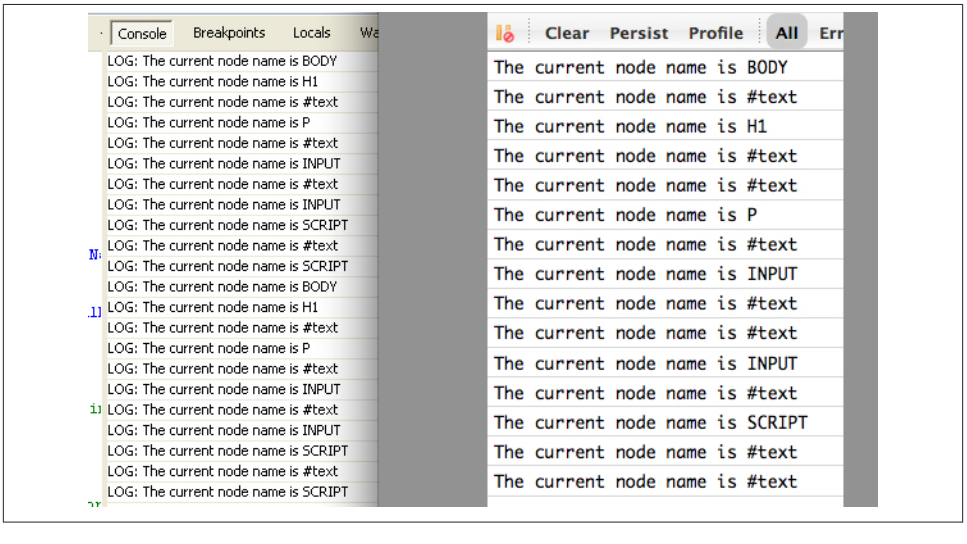


Figure A-20. Traversing the DOM in Internet Explorer and Firefox

Even though some of the output is self-explanatory, there are a number of `#text` nodes that you won't find in the preceding code sample. Unfortunately, web browsers treat

whitespaces differently—some ignore them, whereas others report them as DOM elements. Accordingly, different browsers insert a different number of text nodes in the DOM, representing whitespaces found in the HTML document. So you're better off using one of the JavaScript frameworks for traversing the DOM the cross-browser way. For example, jQuery framework's API for DOM traversing is listed at *http://bit.ly/WXj2r2*.

Styling Web Pages with CSS

CSS stands for *Cascading Style Sheets*. During the past 15 years, several CSS specifications reached the level of Recommendation by W3C: CSS Level 1, 2, and 2.1. The latest CSS Level 3 (a.k.a., CSS3) adds new features to CSS 2.1, module by module, which are listed at the W3C website.

You can find a CSS tutorial as well as tons of other learning resources at webplatform.org.

You can include CSS in a web page by linking to separate files via the HTML tag `<link>`, or by inlining the styles with the tag `<style>`, or by using the `style` attribute in an HTML element (not recommended). For example, if CSS is located in the file *mystyles.css* in the folder *css*, add the following tag to HTML:

```
<link rel="stylesheet" type="text/css" href="css/mystyles.css" media="all">
```

Using the `<link>` tag, you can specify the media where the specific CSS file has to be used. For example, you can have one CSS file for smartphones and another one for tablets. We discuss this in detail in ???.

You should put this tag in the section of your HTML before any JavaScript code to make sure that the styles are loaded before the content of the web page.

Placing the `@import` attribute inside the `<style>` tag allows you to include styles located elsewhere:

```
<style>
   @import url (css/contactus.css)
</style>
```

What's the best way of including CSS in HTML? We recommend using CSS files. Keeping CSS in files separate from HTML and JavaScript makes the code more readable and reusable. You can argue that if your website consists of many files, the web browser will have to make multiple round trips to your server just to load all resources required by the HTML document, which can worsen the responsiveness of your web application.

But usually all files are merged into one before deploying a web application in QA or production servers.

HTML documents are often *prettified* by using CSS class selectors, and you can switch them programmatically with JavaScript. Imagine that a `<style>` section has the following definition of two class selectors, `badStyle` and `niceStyle`:

```
<style>
      .badStyle{
     font-family: Verdana;
     font-size:small;
     color:navy;
     background-color:red;
 }

 .niceStyle{
     font-family: Verdana;
     font-size:large;
     font-style:italic;
     color:gray;
     background-color:green;
 }
</style>
```

Any of these class selectors can be used by one or more HTML elements; for example:

```
<div id="header" class="badStyle">
   <h1>This is my header</h1>
</div>
```

Imagine that an important event has happened and the appearance the `<div>` styled as `badStyle` should programmatically change to `<niceStyle>`. In this case we need to find the `badStyle` element(s) first and change their style. The method `getElementsByClassName()` returns a set of elements that have the specified class name, and because our HTML has only one such element, the JavaScript will use the element zero from this set:

```
document.getElementsByClassName("badStyle")[0].className="niceStyle";
```

The next example illustrates adding a new element to the DOM. Upon clicking a button, the code that follows dynamically creates an instance of type `img` and then assigns the location of the image to its `src` element. In a similar way, we could assign values to any other attributes of the `img` element, including `width`, `height`, or `alt`. The method `appendChild()` is applied to the `<body>` container, but it could be any other container that exists on the DOM:

```
<!DOCTYPE html>
<html>
        <head>
```

```
            <meta charset="utf-8" />
        </head>

    <body>
     <h2>Employee of the month</h2>
        <p>
             <input type="button" value="Show me"
                    onclick="setEmployeeOfTheMonth()"/>
        </p>

     <script>

         function setEmployeeOfTheMonth(){

           // Create an image and add it to the <body> element
           var empImage=document.createElement("img");
                  empImage.setAttribute('src','resources/images/employee.jpg');
                  document.body.appendChild(empImage);
               }

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

Some HTML elements such as `<div>` or `<span>` contain other elements (children), and if you need to change their content, use their property `innerHTML`. For example, to delete the entire content of the document body, just do this: `document.body.innerHTML=""`. You can also use the method `appendChild()`, as shown in the preceding code sample.

If you run this example and click the Show Me button, you'll see an image of the employee of the month added to the `<body>` section of the HTML document, as shown in Figure A-21.

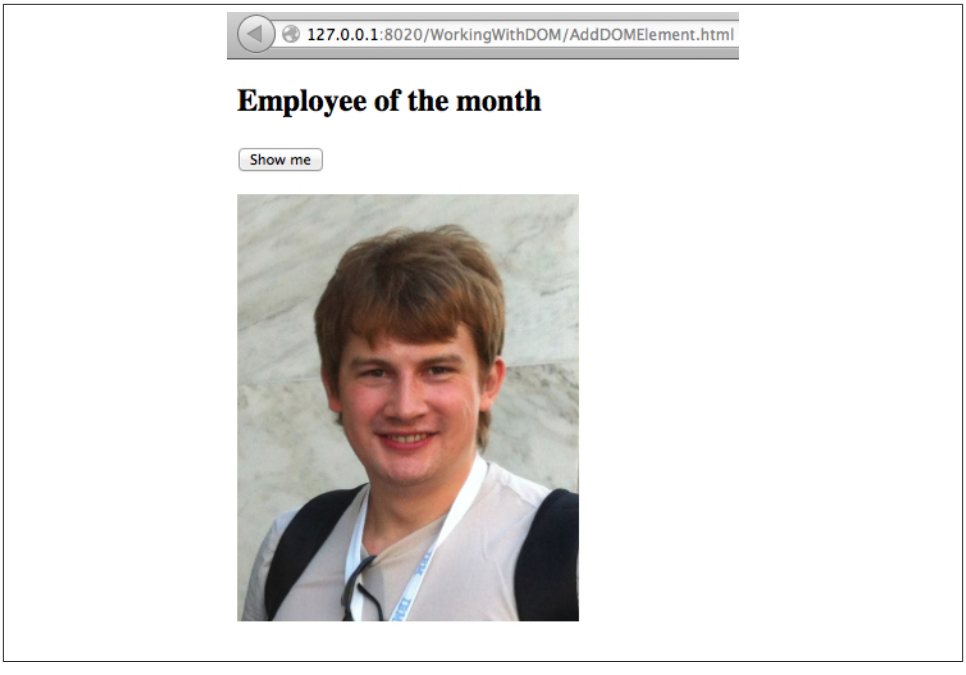


Figure A-21. After clicking the Show Me button

DOM Events

The web browser will notify your application when changes or interactions occur. In such cases, the browser will dispatch an appropriate event (for example, `load`, `unload`, `mousemove`, `click`, and `keydown`). When the web page finishes loading, the browser dispatches the `load` event. When the user clicks a button on a web page, the `click` event is dispatched. A web developer needs to provide JavaScript code that will react to the events important to the application. The browser events will occur regardless of whether you provide the code to handle them. It's important to understand some terms related to event processing.

An *event handler* (a.k.a. *event listener*) is JavaScript code that you want to be called as a response to this event. The last code sample in the previous section was processing the `click` event on the Show Me button as follows: `onclick="setEmployeeOfTheMonth()"`.

Each HTML element has a certain number of predefined *event attributes*, which start with the prefix `on` followed by the name of the event. For example, `onclick` is an event attribute that you can use to specify the handler for the `click` event. You can find out what event attributes are available in the online document "Document Object Model Events."

The preferred way of adding event listeners was introduced in the DOM Level 2 specification back in 2000. You should find the HTML element in the DOM and then assign the event listener to it by calling the method `addEventListener()`. (This is done differently in Internet Explorer earlier than version 9.) For example:

```
document.getElementById("myButton").addEventListener("click",
                                                     setEmployeeOfTheMonth);
```

The advantage of using this programmatic assignment of event listeners is that this can be done for all controls in a central place—for example, in a JavaScript function that runs immediately after the web page completes loading. Another advantage is that you can programmatically remove the event listener if it's no longer needed by invoking `removeEventListener()`. The following example is a rewrite of the last example from the previous section:

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8" />
        </head>

    <body>
     <h2>Employee of the month</h2>
        <p>
             <input type="button" value="Show me" id="myButton"/> ❶
        </p>

     <script>
         window.onload=function(){         // ❷
                document.getElementById("myButton").addEventListener("click",
                                                      setEmployeeOfTheMonth);
         }

         function setEmployeeOfTheMonth(){

           // Create an image and add it to the <body> element
           var empImage=document.createElement("img");
                   empImage.setAttribute('src','resources/images/employee.jpg');
                   document.body.appendChild(empImage);

                 document.getElementById("myButton").removeEventListener("click",
                                                 setEmployeeOfTheMonth); // ❸
                }

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

❶ Compare this button with the one from the previous section: the event handler is removed, but it has an ID now.

❷ When the web page completes loading, a `load` event is dispatched and the function attached to the event attribute `onload` assigns the event handler for the button *click* event. Note that we are passing the callback `setEmployeeOfTheMonth` as the second argument of the `addEventListener()`.

❸ Removing the event listener after the image of the employee of the month has been added. Without this line, each click of the button would add to the web page yet another copy of the same image.

Each event goes through three phases: *capture*, *target*, and *bubble*. It's easier to explain this concept by example. Imagine that a button is located inside `<div>`, which is located inside the `<body>` container. When you click the button, the event travels to the button through all enclosing containers, and this is the capture phase. You can intercept the event at one of these containers even before it reaches the button if need be. For example, your application logic might need to prevent the button from being clicked if a certain condition occurs.

Then, the event reaches the button; this is the target phase. After the event is handled by the button's `click` handler, the event bubbles up through the enclosing containers; this is the bubble phase. You can create listeners and handle this event after the button finishes its processing at the target phase. The next code sample is based on the previous one, but it demonstrates the event processing in all three phases.

Note that if your event handler function is declared by using the event parameter, it will receive the `Event` object (not in Internet Explorer 8), which contains a number of useful parameters. For more information refer, to "Document Object Model Events."

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8" />
        </head>

    <body>
     <h2>Employee of the month</h2>
        <div id="myDiv">
             <input type="button" value="Show me" id="myButton"/>
        </div>

     <script>
         window.onload=function(){
                document.getElementById("myButton").addEventListener("click",
                                                setEmployeeOfTheMonth);

                document.getElementById("myDiv").addEventListener("click",
                                                processDivBefore, true); // ❶
                document.getElementById("myButton").addEventListener("click",
                                                processDivAfter);
```

```
        }

        function setEmployeeOfTheMonth(){

          console.log("Got the click event in target phase");

          // Create an image and add it to the <body> element
          var empImage=document.createElement("img");
                  empImage.setAttribute('src','resources/images/employee.jpg');
                  document.body.appendChild(empImage);

                 document.getElementById("myButton").removeEventListener("click",
                                                    setEmployeeOfTheMonth);
                }

        function processDivBefore(evt){
               console.log("Intercepted the click event in capture phase");

               // Cancel the click event so the button won't get it

               // if (evt.preventDefault) evt.preventDefault();          ❷
               // if (evt.stopPropagation) evt.stopPropagation();
        }

        function processDivAfter(){
               console.log("Got the click event in bubble phase");
        }
    </script>
   </body>
</html>
```

❶ We've added two event handlers on the `<div>` level. The first one intercepts the event in the capture phase. When the third argument of `addEventListener()` is true, this handler kicks in during the capture phase.

❷ If you uncomment these two lines, the default behavior of the `click` event will be cancelled and it won't reach the button at all. Unfortunately, browsers might have different methods, implementing *prevent default* functionality, so additional `if` statements are needed.

Running the preceding example causes the following output in the JavaScript console:

```
Intercepted the click event in capture phase
Got the click event in target phase
Got the click event in bubble phase
```

You can see another example of intercepting the event during the capture phase in ???.

The Microsoft web browsers Internet Explorer 8 and below didn't implement the W3C DOM Level 3 event model; they handled events differently. You can read more on the subject in this MSDN article.

Summary

This appendix covered the JavaScript language constructs that any professional web developer should know. A smaller portion of this appendix illustrated how to combine JavaScript, HTML, and CSS. There are many online resources and books that cover just the HTML markup and CSS, and you'll definitely need to spend more time mastering details of web tools such as Firebug or Google Developer Tools.

Software developers who are coming from strongly typed compiled languages might have a feeling that their productivity drops with JavaScript. We can recommend several medications for this. First, become familiar with the language called CoffeeScript. As respected Java developer James Ward put it, "CoffeeScript is *the* way to write JavaScript." This language is similar to JavaScript and is easy to learn if you understand the JavaScript syntax; it supports classes and is compiled into JavaScript. Visit coffescript.org to see CoffeeScript code snippets and their equivalents in JavaScript.

Another interesting language to learn is Microsoft's TypeScript (it's an open source project). This language is also an extension of JavaScript with added classes, interfaces, and inheritance. It also gets compiled into JavaScript and allows developers to write strongly typed code. TypeScript increases productivity of developers because it helps identify lots of errors related to incorrect types during the compilation phase. TypeScript implements many constructs from ECMAScript 6 and can serve as an example of the JavaScript of the future.

Probably the most interesting new programming language is Google's Dart. This is a compiled language with all object-oriented features: classes, objects, abstract classes, and inheritance. The compiled code runs inside the VM, and Google supports it in the Chrome browser. What about the other browsers? The web application is deployed as a script that automatically checks whether the browser supports Dart. If it does, the compiled code will be sent to the client; otherwise, the Dart code will be automatically compiled into JavaScript, and from the browser's perspective nothing but a JavaScript engine is required. You can perform server-side programming in Dart, too. JetBrains WebStorm, our IDE of choice, supports CoffeeScript, Dart, and TypeScript.