LEARNING JavaScript®

A Hands-On Guide to the Fundamentals of Modern JavaScript

TIM WRIGHT

Praise for *Learning JavaScript*

"Between modern web interfaces, server side technologies, and HTML5 games, JavaScript has never been a more important or versatile tool. To anyone just starting out with JavaScript or looking to deepen their knowledge of the practical core of the language, I would highly recommend *Learning JavaScript*."

—**Evan Burchard**, Independent Web Developer

"Although I've read a couple of books about JavaScript before, as a backend developer, I was thrilled to see Tim Wright's *Learning JavaScript*. The nuances of progressive enhancement versus graceful degradation are finally explained in a manner that someone new to front-end coding can understand. Bravo, Tim."

—**Joe Devon**, Cofounder, StartupDevs.com

"Tim Wright has written a delightfully practical book for the novice front-end developer who wants to learn JavaScript. This book's strength is in providing a good introduction to JavaScript while also illustrating the context of when and where it should be used."

—**R. S. Doiel**, Senior Software Engineer, USC Web Services

"*Learning JavaScript* is a great introduction into modern JavaScript development. From covering the history to its exciting future, Learning JavaScript equips the novice developer to practical application in the workforce. I wish this book came along when I was a novice!"

—**Hillisha Haygood**, Senior Web Developer, Sporting News

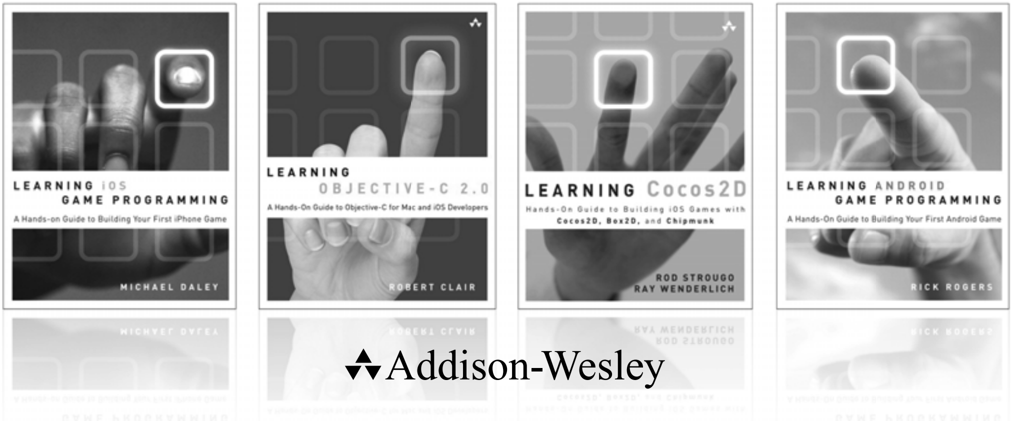
"Tim presents invaluable techniques for writing JavaScript with progressive enhancement at the forefront. If you are new to JavaScript then this book will prove to be a great asset in your learning. Covering all the basics and then right through to touch events, AJAX, and HTML5 APIs, the examples are clear and easy to follow. Using this book, you will learn when and how to use JavaScript to great effect."

—**Tom Leadbetter**, Freelance Web Designer

"*Learning JavaScript* is valuable for both new and veteran developers. It is great for new developers because it is easy to read and provides a step-by-step process to becoming great at JavaScript. Veteran developers will be reminded of many of the best practices they have already forgotten."

—**Christopher Swenor**, Manager of Technology, zMags

Addison-Wesley Learning Series


LEARNING iOS GAME PROGRAMMING
A Hands-on Guide to Building Your First iPhone Game
MICHAEL DALEY
LEARNING OBJECTIVE-C 2.0
A Hands-On Guide to Objective-C for Mac and iOS Developers
ROBERT CLAIR
LEARNING Cocos2D
Hands-On Guide to Building iOS Games with Cocos2D, Box2D, and Chipmunk
ROD STROUGO
RAY WENDERLICH
LEARNING ANDROID GAME PROGRAMMING
A Hands-On Guide to Building Your First Android Game
RICK ROGERS
Addison-Wesley

Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning JavaScript

This page intentionally left blank

Learning JavaScript

A Hands-On Guide to the Fundamentals of Modern JavaScript

Tim Wright

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication data

Wright, Tim, 1982-
Learning JavaScript : a hands-on guide to the fundamentals of modern JavaScript / Tim Wright.
pages cm
Includes bibliographical references and index.
ISBN 978-0-321-83274-0 (pbk. : alk. paper) – ISBN 0-321-83274-4 (pbk. : alk. paper)
1. JavaScript (Computer program language)–Handbooks, manuals, etc. I. Title.
QA76.73.J38W755 2013
005.2'762–dc23
2012019351

ISBN-13: 978-0-321-83274-0
ISBN-10: 0-321-83274-4

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing, August 2012

Editor-in-Chief
Mark Taub

Acquisitions Editor
Laura Lewin

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Barbara Hacha

Indexer
Lisa Stumpf

Proofreader
Debbie Williams

Technical Reviewers
Evan Burchard
Alex Moffat

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
Nonie Ratcliff

❖

For Ma.

❖

Contents

Table of Contents

This page intentionally left blank

Acknowledgments

There are a lot of people who contributed in some way to the completion of this book. First of all, I want to thank the folks at Pearson for giving me the opportunity to not only write this book, but structure it in a way that truly reflects how I believe the topic should be taught. The book would not have stayed on track without them. My technical editors were also instrumental to the process in pointing out any missteps, giving praise when needed, and making sure every detail of the book was written with accuracy and precision; I could not have done it without you (Evan Burchard and Alex Moffat). I would also like to give special thanks to my parents, friends, and family for the continued support, encouragement, and patience throughout this long process and for pulling me out of my “writing cave” for fresh air every once in a while. Without you all, nothing would have been possible.

About the Author

Tim Wright has been a Web designer and front-end developer since 2004, primarily focusing on CSS, HTML5, accessibility, user experience, and building applications with the capability to scale seamlessly from desktop to mobile device. He has worked at various universities nationwide and fostered the advancement of Web standards at each stop along the way.
Tim has written many articles for popular Web design online publications, such as *Smashing Magazine*, *SitePoint*, and *Web Designer Depot*, on all facets of front-end development from HTML5 and CSS3 to user experience and advanced JavaScript techniques. He also writes many articles via his personal blog at csskarma.com. Tim holds a Bachelor's Degree in Marketing Management from Virginia Tech, with a specialization in Graphic Design.

Introduction

When I decided to write a book about JavaScript, I wanted to create it in a way that felt natural to how I learned the language. I didn't learn it from school or a book; my JavaScript knowledge comes from real-world application, trial and error, and self-motivation. I wanted to present the information in a unique way so that you could get up to speed quickly, but still develop a solid base for the language and move forward without feeling overwhelmed with too much information. I combined my teaching experience with how I felt while I was learning to create an environment that moves quickly but has built-in break points and reviews to always keep the mind focused and clear. The JavaScript language can be confusing if taken all at once. There are hundreds of way to accomplish the same task, most of which you don't need to know. I did my best throughout this book to not show too many ways to do the same thing, but rather focus on doing one thing really well.

The organization of this book is a little different from that of a normal JavaScript book. Often terms are introduced, explained in real-time, and readers can feel like they are taking in too much information at once. This can cause a loss of focus on the main task at hand. I addressed this issue by putting all the common JavaScript terms right up front in the book instead of piling them in a glossary that no one will read. As you go through them, they provide brief explanations of many core concepts in the language. This way we don't have to spend valuable time giving broad definitions of miscellaneous terms and can focus on getting you the most knowledge out of this short time we have together.

The process of learning a robust language like JavaScript may seem intimidating at first, but don't worry, it's not that bad. After you grasp some of the basic ideas, the rest is like learning a spoken language; the hard part is properly organizing it, performance tuning, and most of all, knowing when to use CSS instead. Hopefully, by the time you're finished reading this book, you will have gained the knowledge you need to effectively create a better user experience by responsibly using JavaScript.

JavaScript is a language with an amazingly rich history and an even brighter future. Throughout this book you learn the basics of the language, but at the same time you learn more advanced topics, such as HTML5 JavaScript APIs and how you create a touch-enabled interface. You can be assured that even though JavaScript is code, it's far from boring; you can create some pretty wild interfaces and have a lot of fun in the process.

I hope this book can serve you well for years to come and will act as a launching pad for your continued interest in JavaScript. If this is the first step in your journey to learning JavaScript, welcome aboard; if you already know the language, welcome back.

Target Audience for This Book

The audience for this book is anyone starting out in Web design and development who wants to learn about JavaScript. Before reading this book, you should be knowledgeable in HTML and CSS, and be familiar with the concepts behind progressive enhancement.

This book can equally serve absolute beginners and seasoned Web veterans who are expanding their knowledge into JavaScript. All the while, I hope it instills enthusiasm to learn more about this rapidly moving industry.

Code Samples for This Book

The code samples for this book are available on the book's website at
http://learningjsbook.com.

6

Variables, Functions, and Loops

This is one of the more important chapters in the book because you learn some of the core features in JavaScript. We expand on the variables that were mentioned in the previous chapter, then move on to creating functions, and last, we go over how to loop through data to autoexecute the same code block over and over. Using variables, functions, and loops are often the only thing a person knows how to do in JavaScript, and they usually get along just fine. You're already past that part and on your way to becoming an elite JavaScript developer, so no worries there. You'll be coding while all the others are looking up how to do something.

Now that you have a solid base in how to work with a lot of the common things in JavaScript, you can start building an application and producing something tangible. Up to this point in the book, the examples have been pretty specific, but also a little abstract. You've been manipulating content and data, then alerting or observing the result. In this chapter we expand on what you've learned already and begin building a simple JavaScript application that will get more robust as you step through the subsequent chapters.

As you progress though this chapter, you notice that an address book application should be starting to form. Some of the methods that we go over repeat in their core functionality but have very different use-cases. Although they may not necessarily all live in the same application, this is the chapter where you start building that tangible knowledge that can be directly transferred into a project.

Defining Variables

For the most part, you learned about variables within the context of data storage, but they also have an integral part in your application when it comes to functionality.

When considering variable and function naming, it's best to make them meaningful and speak to their contents or purpose. For example, using a variable name of "`myBestFriend`" would be much more helpful than something like, "`firstVariableName`." Something else to consider when naming variables is that they can't start with a number. They can *contain* numbers, such as "`dogs3`" or "`catsStink4Eva`," but they can't begin with a number, such as "`3dogs`."

Grouping Variables

When you're writing an application, it's best to try to group all variables at the top of your JavaScript file or function (when possible) so they can all be immediately cached for later reference. Some people find this method a little unnatural because functions are defined throughout the document, and it's a little easier to maintain when variables are right there with the function they belong to; but grouping variables at the top is one of those small performance boosts you can give to your application. It helps to think of it as one large file containing JavaScript for an application versus thinking of the file as a collection of one-off actions that get executed. When thinking of it as a single unit, it feels a little better (to me) when I'm grouping all variables together at the top.

You can group variables in your document in two ways. Up to this point we have been using a new `var` declaration for each variable; a lot of people prefer this method, and it's perfectly fine to use. An alternative method is to use a single `var` declaration, using commas to separate the individual variables and a semicolon at the very end. Listing 6.1 shows an example of grouping variables with a single `var` declaration. Note the commas at the end of each line.

Listing 6.1 **Grouping Variables with a Single `var` Declaration**

```
var highSchool = "Hill",
    college = "Paul",
    gradSchool = "Vishaal";
```

There's no difference in the way you access these variables compared to how you access variables declared with individual `var` declarations. At the variable level, it's purely a way to group. It isn't good or bad at this point—it's only personal preference. You'll see both methods in looking through JavaScript others have written, so it's good to know what's going on.

You see this style of variable declaration a lot more when getting into objects, methods, and grouping functions together. I prefer it because it feels cleaner and a little more consistent, but as you progress you will settle on a preference of your own. Both are certainly valid methods.

Reserved Terms

JavaScript contains a lot of core functionality. We've been over quite a bit of it so far. Beyond that core functionality you will be defining a lot of your own custom code. If the names of your custom JavaScript match up with anything built into the language, it can cause collisions and throw errors. It's the same as if you're writing a large JavaScript file—you want to make sure all the function and variable names are as unique as possible to prevent problems and confusion while parsing the information. If you have two functions with the same name, it's difficult to tell the browser which one to use, so it's just not allowed.

To prevent these issues with native JavaScript, there are some reserved words (keywords) that you can't use when defining variables, functions, methods, or identifiers within your code. Following is a list of the reserved words:

- `break`
- `case`
- `catch`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `else`
- `finally`
- `for`
- `function`
- `if`
- `implements`
- `in`
- `instanceof`
- `interface`
- `new`
- `package`
- `private`
- `protected`
- `public`
- `return`
- `static`
- `switch`
- `this`
- `throw`
- `try`
- `typeof`
- `var`
- `void`
- `while`
- `with`

Most of these are no-brainers, like `function` and `var`, and under normal circumstances you probably would never come across a situation where something like "`implements`" would be a reasonable name for a variable or function. If you end up using any of these terms in your code, the console will throw an error and let you know that you're using a reserved word. With that in mind, I think the value in this list is not so much memorizing it, but rather recognizing that these words map to native actions in the language. It will help you write better code and also aid in learning more advanced JavaScript down the road if you choose to research some of those terms that are beyond the scope of this book, such as public, private, and protected.

Functions

Functions in any programming language are ways to write code that can be used later. At its most basic form, this is also true for JavaScript. You can write a chunk of custom code and not only execute it at will, but you can also execute it over and over, which can help streamline your application by increasing its maintainability (declaring a chunk of code one time and referencing it, rather than rewriting what it does). It's like keeping all your CSS in the same file or why you keep all JavaScript in the same file—you know exactly where it is when you need to change or add something.

You've been using functions already in earlier chapters when you pass data into an `alert()`. "Alert" is technically called a `method` but for all intents and purposes, it's the same as a function.

Basic Functions

The chance of creating a JavaScript application without having to write your own functions is pretty low. It's something that you'll be doing on every project, and it's very easy to do using the function keyword (remember the reserved words list? This is what `function` is for).

Using the function keyword is like saying, "Hey, I'm building something over here that should be treated as a function." Listing 6.2 shows a basic function declaration.

Listing 6.2 **Writing a Basic Function**

```
function sayHello() {

    alert("hey there! ");

}
```

Calling a Function

Calling a function is very simple. You type out the name, and then parentheses and the function will be executed. The parentheses tell the browser that you want to execute the function and to use any data (arguments) contained within the parentheses within the function. Listing 6.2.1 shows how to call the function we declared in Listing 6.2. It should alert the text, "hey there!"

Listing 6.2.1 **Calling a Basic Function**

```
sayHello(); // hey there
```

Arguments

Arguments are a way to pass information or data into a function. As previously mentioned, up to this point you've been using the `alert()` method. We've also been passing it arguments. The alert method is designed in native JavaScript to take arguments and display them in the form of a pop-up box in the browser.

Functions can take any number of arguments. They can be any type of information; strings, variables, large data sets, and anything else you can think of can be passed into a function through an argument. As you're defining your functions, you will be assigning names to the arguments, sort of like the way you assign names to a variable. After that argument is named in the function, it becomes a variable you'll be using inside that function.

In Listing 6.2.2 you can see that the `sayHello()` function now has a single argument called "message." Inside, the function "message" is used as a variable that gets passed into the JavaScript `alert()` method.

Listing 6.2.2 **Passing a Function Variable Through Arguments**

```
/* declare the function */
function sayHello(message){

    alert(message); // "message" is also an argument in the "alert" method

}

/* call it a couple times with different messages */
sayHello("Hey there, you stink!");

sayHello("I feel bad I just said that.");
```

When this function is called, we're setting the string argument to "Hey there, you stink!" and then quickly apologizing with another alert, because frankly it was kind of rude. This is a very real-life way arguments are used in functions. The string can either be declared upon calling the function (like we're doing in Listing 6.2.2) or it can be declared immediately in the function declaration. (Instead of using the message variable, you could insert the string.) Calling it the way we did is much more common in the real world, though.

Anonymous Functions

Anonymous functions are functions that have no name (obviously—they're anonymous). They execute immediately and can contain any number of other functions. The syntax for declaring an anonymous function is a little different. They are dynamic in nature because they are executed at runtime rather than waiting to be called.

Anonymous functions perform very well in the browser because there is no reference to them in another part of the document. This comes with pluses and minuses. So as you write your JavaScript, it is always good to note that if you have to rewrite an anonymous function over and over, it's probably best to pull it out into a normal function to cut down on maintenance and repetitive code.

There is often a little confusion as to the purpose of anonymous functions. If you want something to execute at runtime, why wouldn't you just dump the code right into your JavaScript file? Why even bother wrapping it in an anonymous function? Well, this is a good place to bring up a term you may hear a lot: **scope**.

Scope

Scope is a programming concept that exists to reduce the amount of variable and function collisions in your code. It controls how far information can travel throughout your JavaScript document. Earlier on, we briefly mentioned `global variables`. "Global" is a type of scope; the global scope for a variable means that the variable can be accessed and used anywhere in the document. Global variables are generally a bad thing, especially in larger files where naming collisions are more likely. Try to keep things out of the global scope if possible. Listing 6.3 shows how to declare a basic anonymous function and keep variables out of the global scope.

Listing 6.3 **Defining an Anonymous Function**

```
/* set up your anonymous function */
(function () {

    /* define a variable inside the function */
     var greeting = "Hello Tim";

     /* access the variable inside the function */
     alert("in scope: " + greeting);

})(); // end anonymous function
```

For the most part, you will be dealing in function-level scope. This means that any variable defined inside a function cannot be used outside that function. This is a great benefit of using anonymous functions. If you wrap a code block in an anonymous function, the contents of that function, which would normally default to the global scope, will now be contained within the scope of that anonymous function.

Listing 6.3.1 defines a variable inside an anonymous function, alerts the variable, and then tries to alert the variable again, outside the function (it won't end well).

Listing 6.3.1 **Showing Scope Inside an Anonymous Function**

```
/* set up your anonymous function */
(function () {

    /* define a variable inside the function */
     var greeting = "Hello Tim";
     /* access the variable inside the function */
     alert("in scope: " + greeting);

})(); // end anonymous function

/* try and access that variable outside the function scope */
alert("out of scope: " + typeof(greeting)); // alerts "undefined"
```

As you can see, the variable `alert` is `undefined`, even though you can see it's clearly defined within the anonymous function. This is because the function scope will not allow the variable to leave the function.

> **Note**
>
> In the second alert of Listing 6.3.1 we're using the JavaScript method `typeof()`, which alerts the variable type "undefined." If we didn't do this, the file would throw an error, and you wouldn't see the second alert at all. The JavaScript console would display the error, "greeting is undefined."

Calling a Function with a Function

When you have a function that calls another function, the second function is referred to as a **callback**. The callback function is defined as a normal function with all the others but is executed inside another function. They're a little different because instead of *you* having to do something to execute the function, another function does something. It's like having robots that are built by other robots—total madness, I know.

Callback functions are a great way to separate out the levels of functionality in your code and make parts more reusable. Often you will see callback functions passed as arguments to other functions. We'll get more into that in the next chapter when we talk about JavaScript events, and they're especially important when dealing with server communications like Ajax. Listing 6.3.2 shows our `sayHello()` function being defined and then called inside the anonymous function. In this case, `sayHello()` is a callback function (calling it twice).

Listing 6.3.2 **Using a Callback Function**

```
function sayHello(message) {
    alert(message);
}

(function (){

    var greeting = "Welcome",
        exitStatement = "ok, please leave.";
    sayHello(greeting);
    sayHello(exitStatement);

})();
```

Returning Data

Every function you create will not result in a direct output. Up to this point you've been creating functions that do something tangible, usually alerting a piece of data into the browser. You won't always want to do that, though; from time to time you will want to create a function that returns information for another function to use. This will make your functions a little smaller, and if the function that gathers information is general enough, you can reuse it to pass the same (or different) information into multiple functions.

Being able to return data and pass it into another function is a powerful feature of JavaScript.

Returning a Single Value

Going back to the `sayHello()` function that was defined in Listing 6.2, we're going to remove the `alert()` action that was previously being executed when the function was called, and we'll replace it with a return statement. This is depicted in Listing 6.3.3.

Listing 6.3.3 **Returning Data with a Function**

```
function sayHello(message){
    return message + "!"; // add some emotion too
}
```

You'll probably notice that the `sayHello()` function doesn't do anything in the browser anymore. That's a good thing (unless you're getting an error—that's a bad thing). It means the function is now returning the data but it's just sitting there waiting to be used by another function.

Returning Multiple Values

Sometimes returning a single value isn't enough for what you're trying to accomplish. In that case you can return multiple values and pass them in an array format to other functions. Remember how I mentioned that arrays are really important? They creep up a lot when dealing in data storage and flow in JavaScript. In Listing 6.3.4 you can see the `sayHello()` function taking two arguments. Those arguments get changed slightly and are resaved to variables; then they are returned in an array format to be accessed later.

Listing 6.3.4 **Returning Multiple Data Values with a Function**

```
function sayHello(greeting, exitStatement){

    /* add some passion to these dry arguments */
    var newGreeting = greeting + "!",
        newExitStatement = exitStatement + "!!";

    /* return the arguments in an array */
    return [newGreeting, newExitStatement];

}
```

Passing Returned Values to Another Function

Now that you're returning variables, the next step is to pass those variables into another function so they can actually be used. Listing 6.3.5 shows the `sayHello()` function from Listing 6.3.1 returning an array of information and a new function called `startle()`, taking two arguments, passing them through the original `sayHello()` function, and alerting the results.

Listing 6.3.5 **Using Returned Function Values Passed into Another Function**

```
function sayHello(greeting, exitStatement){

    /* add some passion to these dry arguments */
    var newGreeting = greeting + "!",
        newExitStatement = exitStatement + "!!";

    /* return the arguments in an array */
    return [newGreeting, newExitStatement];

}

function startle(polite, rude){

     /* call the sayHello function, with arguments and same each response to a
➥variable */
    var greeting = sayHello(polite, rude)[0],
        exit = sayHello(polite, rude)[1];

    /* alert the variables that have been passed through each function */
    alert(greeting + " -- " + exit);

}

/* call the function with our arguments defined */
startle("thank you", "you stink");
```

A Function as a Method

Just as you can group variables and data into objects, you can also do it with functions. When you group functions into objects, they're not called functions anymore; they're called "methods."

When I first started out with JavaScript, I came in from a design background rather than as a developer. This meant that I wasn't familiar with common programming terms such as object, function, method, loop, and so on. I quickly learned what a function was and how to work with them through a lot of Googling. But I would hear people talk about the `alert()` method and other methods native to JavaScript, and I wouldn't really get it because they look the same as functions. Why isn't it the "alert function"? I had no idea. This comes up a lot when you're dealing with JavaScript libraries as well (we get into that later in the book); everything is a method and nothing is a function, even though they all look and act the same.

Here's what's going on. In Chapter 5, "Storing Data in JavaScript," you learned about storing information in objects. I mentioned that you could also store functions in objects. When you do that, they're called methods instead of functions, but they work the same way. It's weird, I know, and it's not even an important distinction while you're coding. It's more about organizing your functions in groups to make them easier to maintain. The `alert()` method lives inside a global object (you never see it), which is why it's called a method.

Now that we're past that ordeal, organizing your functions into meaningful objects can clean up a lot of your code, especially on larger projects where you need the code organization help to keep your sanity. Listing 6.4 should look a little familiar; it shows how to organize our two functions (`sayHello` and `startle`) inside an object called "`addressBookMethods`." If we were building a large-scale application with many features, this would be a great way to section off the functionality meant only for the address book feature.

Listing 6.4 **Grouping Similar Functions**

```
var addressBookMethods = {

    sayHello: function(message){

        return message;

    },
    startle: function(){

        alert(addressBookMethods.sayHello("hey there, called from a method"));

    }

}

/* call the function */
addressBookMethods.startle();
```

Calling a method is a little different from calling a function. You'll notice in Listing 6.4 that instead of calling `startle()` by itself, you have to call `addressBookMethods.startle().` This is because before you can access the method, you have to access the object and drill down to the method.

Performance Considerations

Nesting functions in objects has the same performance implications that we spoke of when nesting variables in objects. The deeper a function is nested inside an object (`addressBookMethods`), the more resources it takes to extract. This is another place in your code where you will have to balance performance with maintainability. We're not talking a ton of time here—maybe a few milliseconds difference—but it can add up. Most of the time it won't matter, but if you find yourself needing a performance boost, function objects would be a place to look for a bottleneck. I probably wouldn't go more than a few levels deep when creating these objects. Listing 6.4 goes only one level deep, which is a nice balance between performance and maintainability.

Loops

A loop will execute a block of code over and over until you tell it to stop. It can iterate through data or HTML. For our purposes we'll mostly be looping through data. Much the way a function is a chunk of JavaScript code, a loop can make that function execute over and over—like a little buddy you have to do your repetitive tasks for you. And they're built right into the language!

For this one, we need some data to loop through. We'll be using contact information for the data and saving it to a JSON object called "contacts." Listing 6.5 shows a small sample of the contact information we'll be looping through. I find it easier to work with data that represents people, because when something goes wrong with one of the items it's more difficult to get angry at someone you know than it is at anything else. Feel free to substitute your own friends or family in the data so you don't get frustrated if something goes wrong.

Listing 6.5 **Creating Data in a JSON Object**

```
var contacts = {
    "addressBook" : [
        {
            "name": "hillisha",
            "email": "hill@example.com",
        },
        {
            "name": "paul",
            "email": "cleveland@example.com",
        },
        {
            "name": "vishaal",
            "email": "vish@example.com",
        },
        {
            "name": "mike",
            "email": "grady@example.com",
        },
        {
            "name": "jamie",
            "email": "dusted@example.com",
        }
    ]
};
```

for Loop

There are few different types of loops in JavaScript, a `while` loop, a `do-while` loop, and a `for` loop. Most of them are perfectly fine; I would avoid the `foreach` loop because it's known to be a performance hog, but the others are fine to use. A `while` loop and a `for` loop are basically the same thing, but the `for` loop is a little more direct, to the point, and it's the most common kind of loop you're going to find in the wild. In all the years I've been writing JavaScript, it's been 99% `for` loops. With that in mind, we're going to go over the `for` loop in this book.

Listing 6.5.1 will show you a basic `for` loop, and then we'll go over what's happening.

Listing 6.5.1 **A `for` Loop Integrating Address Book Data**

```
/* cache some initial variables */
var object = contacts.addressBook,
    contactsCount = object.length,
    i;

/* loop through each JSON object item until you hit #5, then stop */
for (i = 0; i < contactsCount; i = i + 1) {

     // code you want to execute over and over again

} // end for loop
```

Right away, you can see that we're saving some information to variables. The first variable "`object`" is saving the JSON object we create to a variable so it's a little easier to work with. The second variable, "`contactsCount`", looks through the JSON object and counts the number of items in there. This will let us know how many times to loop through the data. The third variable, "`i`", is just a way to declare the counting variable for our loop. Later on we'll be setting the value.

Inside the for you can see three statements. The first statement is setting the counter variable (i) to its initial value of 0 (we start at 0). The second statement is the condition in which you run the loop. As long as the "i" value is less than the overall count of items in the data, it should execute the code contained inside the loop brackets { }. The last statement takes the "i" value and adds 1 to it each time the loop executes until it's no longer less than the overall count. In our case, this loop will execute 5 times because there are five people in the address book.

Listing 6.5.2 will show the actual loop to iterate through the address book data saved to the JSON object, and then, using the innerHTML DOM method, output the result into the document's <body> element. Besides the output, a main difference to note in Listing 6.5.2 is that we're now running a check on the contactsCount variable to make sure it's greater than 0 before continuing onto the loop. This is a general best practice to prevent unnecessary code from executing should there be an error with the data.

Listing 6.5.2 **A for Loop Integrating Address Book Data**

```
/* cache some initial variables */
var object = contacts.addressBook, // save the data object
    contactsCount = object.length, // how many items in the JSON object? "5"
    target = document.getElementsByTagName("body")[0], // where you're outputting the
➥data
    i; // declare the "i" variable for later use in the loop

/* before doing anything make sure there are contacts to loop through */
if(contactsCount > 0) {

    /* loop through each JSON object item until you hit #5, then stop */
    for (i = 0; i < contactsCount; i = i + 1) {

        /* inside the loop "i" is the array index */
        var item = object[i],
            name = item.name,
            email = item.email;

        /* insert each person's name & mailto link in the HTML */
       target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

    }
}
```

It's nice to be rid of that annoying alert box, isn't it? Rather than alerting each value, we are now choosing a target within the HTML document (<body> element) and outputting the data there. This is more along the lines of what you'll be doing in the real world, so we'll be doing that now instead of using the alert() method.

Performance Considerations

As mentioned in an earlier chapter, JavaScript, by nature, is blocking. That means it will stop the download of other objects on the page until it is finished with its business. This can be very evident when dealing with loops. The data we're dealing with here is only five items in length, so there isn't a problem executing this block of code 5 times. However, as the number of elements you're looping through increases, so will the time it takes to iterate over them. This is important to note when you're looping through a lot of items because it can really bog down the loading time of a page.

Any variable that doesn't change and can be defined outside the loop *should* be defined outside the loop. You'll notice in our loop that there is a variable called `contactsCount`; it is defined outside the loop and then referenced within. We can do this because the length of the data never changes while the information is being looped through. If it were inside the loop, the length would have to be recalculated each time the loop ran, which can get very resource intensive. Little things like that can help you conserve resources when you're working with loops.

Conditionals

Conditionals are how you let your program make decisions for you. Decisions can be based on the data presented (decisions you make) or based on user input, like one of those choose-your-own adventure books. It's a way to inject some logic into your JavaScript.

Conditionals can be used for everything from outputting different information into the DOM to loading a completely different JavaScript file. They're very powerful things to have in your JavaScript toolkit.

if Statement

By far, the most common type of conditional is the `if` statement. An `if` statement checks a certain condition, and if true, executes a block of code. The `if` statement is contained within two curly brackets `{ }`, just like the loops we were talking about earlier and the functions before that.

This is best described through a coding sample so let's move right to it. In Listing 6.5.3 you can see a basic `if` statement that is being applied inside the loop of our JSON object in Listing 6.5.2. Inside the loop, if the person's name is "hillisha" the name and mailto link with an exclamation point at the end will be outputting into the document. This output should only be Hillisha's `mailto` link without any other names.

Listing 6.5.3 **Basic `if` Statement**

```
/* if "hillisha comes up, add an exclamation point to the end" */

if(name === "hillisha"){

    target.innerHTML += '<p><a href="mailto:' + email + '">' + name + '</a>!</p>';

}
```

> **Note**
>
> Note that we're using "===" in the conditional to check if the names match what we're looking for. This triple equal sign operator signifies an *exact* match. There is also a double equal sign (==) you can use that means "match." It's best practice to use === rather than == because it's more specific, and when dealing in Boolean values it can get confusing because true = 1 and false = 0. Therefore if you're looking for a "false" Boolean value, using a double equal sign would not only return what you're looking for, but a "0" would do the same. In a nutshell, **use the === operator and not the == operator** and you won't hit that weird gray area of false versus 0 and true versus 1 when dealing with Booleans.

if/else Statement

In Listing 6.5.3 the output was only a single person's name because the condition was set to handle only that one instance of `name === "hillisha".` Normally you will want do something for the rest of the people in your address book as they are outputted. The `if/else` statement is for just that purpose.

The `if/else` statement gives you the capability to create multiple conditions and then a fallback condition for any items that don't meet the conditions' criteria. In Listing 6.5.4 you can see that we are still looping through the address book JSON object, but this time we're setting three conditions:

- if name is hillisha
- if name is paul
- everyone else

Listing 6.5.4 **`if/else` Statement**

```
if(name === "hillisha"){

    /* if "hillisha comes up, add an exclamation point to the end" */
    target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a>!</p>';
} else if (name === "paul") { // line 5

    /* if "paul" comes up, add a question mark */
    target.innerHTML += '<p><a href="mailto:' + email + '">' + name + '</a>?</p>';

/* otherwise, output the person as normal*/
} else {

     target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

}
```

On line 5 in Listing 6.5.4, you can see that you can combine the two types of statements into `else if` to create a flow of conditional statements. Using this method, there is no limit to the amount of conditionals you can write. When you get to a large number of conditionals like this, you may consider changing from an `if/else` statement to a slightly more efficient `switch` statement.

switch Statement

A `switch` statement, on the surface, functions almost exactly like an `if/else` statement. In a `switch` statement, you first have to set a switch value (the thing you're going to check for); in this example, we have been checking for `name`, so that's the switch value. You then set up cases to test against. We checked for "hillisha" once and also "paul"; those would be the cases used. Last, there is a default state if none of the cases return as true.

The `switch` statement in Listing 6.5.5 creates the same output as the `if/else` statement in Listing 6.5.4, but under the hood and in syntax they are pretty different. Let's take a look at this `switch` statement.

Listing 6.5.5 **Basic `switch` Statement**

```
switch(name){
    case "hillisha":

        /* if "hillisha comes up, add an exclamation point to the end" */
        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a>!</p>';

        /* break out of the statement */
        break;

    case "paul":

        /* if "paul" comes up, add a question mark */
        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a>?</p>';
        /* break out of the statement */
        break;

    default:

        /* otherwise, output the people as normal*/
        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

} // end switch statement
```

if versus switch

Besides syntax there is one major difference in how an `if/else` statement functions when compared to a `switch` statement. First, the `else` in an `if/else` isn't required; you can just run an `if` statement like we did in Listing 6.5.3. In a `switch` statement, the default option is required.

The iteration mechanism is also different. In the `if/else` statement in Listing 6.5.4, it still runs the same process over each item in the JSON object. For example, the first person listed is "Hillisha," so when the conditional statement is executed on that item, it asks three questions:

- Does this name equal "hillisha?" – `true`
- Does this name equal "paul?" – `false`, it's "hillisha"
- Does it equal something else – `false`

Even if the first condition is true, the statement continues checking against the other conditions. If you have a lot of conditions, this can be very resource intensive. This is where the `switch` statement really shines.

In the `switch` statement, after a condition is found to be true, it breaks out of the cases so there are no more checks made. In the `switch` statement in Listing 6.5.5, the second condition of looking for the name "paul" would look something like this:

- Does this name equal "paul?" – `false`, it's "hillisha"
- Does this name equal "paul?" – `true`, found it!
- Stop asking questions you know the answer to.

Many people like using `if/else` because it feels more natural, but after you get to a certain conditional count, you should consider moving over to the `switch` statement for a little better performance in your JavaScript.

Putting It All Together

Up to this point in the chapter, you have been building a simple address book and outputting the data.

Listing 6.6 is a cumulative dump of the code you’ve been putting together. It contains the JSON object with contact information, an anonymous function, and a loop with a conditional statement to check the JSON object length.

Listing 6.6 **Application Code**

```
/* create some data in the form of a JSON object you can consume and loop through */

var contacts = {
    "addressBook" : [
        {
            "name": "hillisha",
            "email": "hill@example.com",
        },
        {
            "name": "paul",
            "email": "cleveland@example.com",
        },
        {
            "name": "vishaal",
            "email": "vish@example.com",
        },
        {
            "name": "mike",
            "email": "grady@example.com",
        },
        {
            "name": "jamie",
            "email": "dusted@example.com",
        }
    ]
};

/* wrap everything in an anonymous function to contain the variables */
(function () {

/* cache some initial variables */

var object = contacts.addressBook, // save the JSON object
    contactsCount = object.length, // how many items in the JSON object? "5"
    target = document.getElementsByTagName("body")[0], // where you're outputting the
➥data
```

```
     i; // declare the "i" variable for later use in the loop

/* before doing anything make sure there are contacts to loop through */

if(contactsCount > 0) {

    /* loop through each JSON object item until you hit #5, then stop */

    for (i = 0; i < contactsCount; i = i + 1) {

        /* inside the loop "i" is the array index */

        var item = object[i],
            name = item.name,
            email = item.email;

        target.innerHTML += '<p><a href="mailto:'+ email +'">' + name + '</a></p>';

    } // end for loop

} // end count check

})(); // end anonymous function
```

There's the address book application as it stands right now. You've created the contact information for our five friends and inserted them into a JSON object. After storing the JSON object, you're looping through each item (person) and outputting them individually into the <body> element, one after another. You're also creating HTML fragments that are paragraphs and mailto links for each person.

The processes of looping through data, storing the items as variables, and outputting them into the DOM is, by far, the most common looping method you will see as you build more applications with JavaScript. This code will not only serve as a base for our application, but as a good reference point for your future JavaScript development.

Summary

In this chapter, we started off by diving a little deeper into variables. You learned the different grouping options when declaring variables, along with some best practice considerations like why you should declare variables at the top of your JavaScript document. We also went over the list of reserved terms you should consider when naming functions and variables to help prevent collisions in your scripting file.

After that, we elaborated on the different types of functions, how they differ from each other, and discussed different case scenarios for when you might want to use each type of function.

We talked about basic functions, anonymous functions, callback functions, and functions in objects, along with how to get your functions working together by returning and passing data to one another, returning both single and multiple values.

Before this chapter, we were accessing items directly when working with data. This chapter showed how to execute the same code over and over for each data item in the form of a loop. We learned about the `for` loop specifically and talked about performance considerations and why the loop is assembled in the way it is.

After loops, we got into conditionals in the form of `if/else` and `switch` statements. They appear similar on the surface, but we also talked about why they're different and the scenarios where you may want to use one style over the other.

This chapter was the first step in building a real JavaScript application (an address book). In the next chapter, we start to bring users into the mix when we talk about events, how we might apply user interactions to this application, and learn some general information about events in JavaScript.

Exercises

1. Why is it best to position all variables at the top of your JavaScript file?
2. Why are some words reserved in JavaScript?
3. How are anonymous functions different from basic functions?

Index