Phil Ballard

Sixth Edition
New coverage of ECMAScript 6

Sams **Teach Yourself**

JavaScript™

in **24** **Hours**

SAMS

About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

Sams Teach Yourself JavaScript® in 24 Hours

Sixth Edition

Phil Ballard

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself JavaScript in 24 Hours, Sixth Edition

Copyright © 2015 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33738-3

ISBN-10: 0-672-33738-X

Library of Congress Control Number: 2015905614

Printed in the United States of America

First Printing June 2015

Executive Editor
Mark Taber

Managing Editor
Sandra Schroeder

Senior Development Editor
Chris Zahn

Senior Project Editor
Tonya Simpson

Copy Editor
Bart Reed

Indexer
Tim Wright

Proofreader
Debbie Williams

Publishing Coordinator
Vanessa Evans

Technical Editor
Siddhartha Singh

Cover Designer
Mark Shirar

Compositor
Bronkella Publishing

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Contents at a Glance

Table of Contents

About the Author

Phil Ballard, the author of various *Sams Teach Yourself* titles, graduated in 1980 with an honors degree in electronics from the University of Leeds, England. Following an early career as a research scientist with a major multinational, he spent a few years in commercial and managerial roles within the high technology sector, later working full time as a software engineering consultant.

Operating as "The Mouse Whisperer" (www.mousewhisperer.co.uk), Ballard has spent recent years involved solely in website and intranet design and development for an international portfolio of clients, as well as writing numerous technical books and articles.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

This introduction walks you through a few basic things before you begin reading, including who this book was written for, why it was written, the conventions employed in this book and in the Sams Teach Yourself series, how the content is organized, and the tools you need to create JavaScript.

Who This Book Is For

If you're interested in learning JavaScript, chances are that you've already gained at least a basic understanding of HTML and web page design in general, and want to move on to adding some extra interactivity to your pages. Or maybe you currently code in another programming language, and want to see what additional capabilities JavaScript can add to your armory.

If you've never tinkered with HTML at all, nor done any computer programming, it would be helpful to browse through an HTML primer before getting into the book. Don't worry—HTML is very accessible, and you don't need to be an expert in it to start experimenting with the JavaScript examples in this book.

JavaScript is an ideal language to use for your first steps in programming, and in case you get bitten by the bug, pretty much all of the fundamental concepts that you learn in JavaScript will later be applicable in a wide variety of other languages such as C, Java, and PHP.

The Aims of This Book

When JavaScript was first introduced, it was somewhat limited in what it could do. With basic features and rather haphazard browser support, it gained a reputation in some quarters as being something of a toy or gimmick. Now, due to much better browser support for W3C standards and improvement in the JavaScript implementations used in recent browsers, JavaScript is finally being treated as a serious programming language.

Many advanced programming disciplines used in other programming languages can readily be applied to JavaScript; for example, object-oriented programming promotes the writing of solid, readable, maintainable, and reusable code.

So-called "unobtrusive" scripting techniques and the use of DOM scripting focus on adding interaction to web pages while keeping the HTML simple to read and well separated from the program code.

This book aims to teach the fundamental skills relevant to all of the important aspects of JavaScript as it's used today. In the course of the book, you start from basic concepts and gradually learn the best practices for writing JavaScript programs in accordance with current web standards.

Conventions Used

All of the code examples in the book are written as HTML5. For the most part, though, the code avoids using HTML5-specific syntax, since at the time of writing its support in web browsers is still not universal. The code examples should work correctly in virtually any recent web browser, regardless of the type of computer or operating system.

In addition to the main text of each lesson, you will find a number of boxes labeled as Notes, Tips, and Cautions.

Note

These sections provide additional comments that might help you to understand the text and examples.

Tip

These blocks give additional hints, shortcuts, or workarounds to make coding easier.

Caution

Avoid common pitfalls by using the information in these blocks.

Try it Yourself

Each hour contains at least one section that walks you through the process of implementing your own script. This will help you to gain confidence in writing your own JavaScript code based on the techniques you've learned.

Q&A, Workshop, and Exercises

After each hour's lesson, you'll find three final sections.

- The "Q&A" section tries to answer a few of the more common questions about the hour's topic.
- The "Workshop" section includes a quiz that tests your knowledge of what you learned in that lesson. Answers to the quiz items are conveniently provided immediately following the quiz.
- The "Exercises" section offers suggestions for further experimentation, based on

the lesson, that you might like to try on your own.

How the Book Is Organized

The book is divided into seven parts, gradually increasing in the complexity of the techniques taught.

- **Part I—First Steps with JavaScript**

  An introduction to the JavaScript language and how to write simple scripts using the language's common functions. This part of the book is aimed mainly at readers with little or no prior programming knowledge, and no knowledge of the JavaScript language.

- **Part II—Cooking with Code**

  Here JavaScript's data types are introduced, such as numbers, strings, and arrays. More sophisticated programming paradigms such as program control loops and timers are also covered.

- **Part III—Objects**

  This part of the book concentrates on creating and handling objects, including navigating and editing the objects belonging to the DOM (Document Object Model).

- **Part IV—HTML and CSS**

  Here you learn in greater depth how JavaScript can interact with HTML (including HTML5) and CSS (Cascading Style Sheets), including the latest CSS3 specification.

- **Part V—Using JavaScript Libraries**

  In this part of the book you learn how to simplify cross-browser development using third-party libraries such as jQuery.

- **Part VI—Advanced Topics**

  This part of the book covers reading and writing cookies, looks at what's new in JavaScript via the ECMAScript 6 specification, introduces the use of frameworks such as AngularJS, and shows examples of using JavaScript beyond its use in web pages.

- **Part VII—Learning the Trade**

  In the final part you explore aspects of professional JavaScript development such as good coding practices, JavaScript debugging, and unit testing.

Tools You'll Need

Writing JavaScript does not require any expensive and complicated tools such as

Integrated Development Environments (IDEs), compilers, or debuggers.

The examples in this book can all be created in a text-editing program, such as the Windows Notepad program. At least one such application ships with just about every operating system, and countless more are available for no or low cost via download from the Internet.

Note

Appendix A, "Tools for JavaScript Development," lists some additional, easily obtainable tools and resources for use in JavaScript development.

To see your program code working, you'll need a web browser such as Internet Explorer, Mozilla Firefox, Opera, Safari, or Google Chrome. It is recommended that you upgrade your browser to the latest current stable version.

The vast majority of the book's examples do not need an Internet connection to function. Simply storing the source code file in a convenient location on your computer and opening it with your chosen browser is generally sufficient. The exceptions to this are the hour on cookies and the examples in the book that demonstrate Ajax; to explore all of the sample code will require a web connection (or a connection to a web server on your local area network) and a little web space in which to post the sample code. If you've done some HTML coding, you may already have that covered; if not, a hobby-grade web hosting account costs very little and will be more than adequate for trying out the examples in this book. (Check that your web host allows you to run scripts written in the PHP language if you want to try out the Ajax examples in Part V. Nearly all hosts do.)

Part I: First Steps with JavaScript

Hour 1. Introducing JavaScript

What You'll Learn in This Hour:

- About server-side and client-side programming
- How JavaScript can improve your web pages
- The history of JavaScript
- The basics of the Document Object Model (DOM)
- What the `window` and `document` objects are
- How to add content to your web pages using JavaScript
- How to alert the user with a dialog box

The modern Web has little to do with its original, text-only ancestor. Modern web pages can involve audio, video, animated graphics, interactive navigation, and much more—and more often than not, JavaScript plays a big part in making it all possible.

In this first hour we describe what JavaScript is, briefly review the language's origins, and consider the kinds of things it can do to improve your web pages. You also dive right in and write some working JavaScript code.

Web Scripting Fundamentals

Since you've picked up this book, there's a pretty good chance that you're already familiar with using the World Wide Web and have at least a basic understanding of writing web pages in some variant of HTML.

HTML (Hypertext Markup Language) is not a *programming* language but (as the name indicates) a *markup* language; we can use it to mark parts of our page to indicate to the browser that these parts should be shown in a particular way—bold or italic text, for instance, or as a heading, a list of bullet points, arranged as a table of data, or using many other markup options.

Once written, these pages by their nature are *static*. They can't respond to user actions, make decisions, or modify the display of their page elements. The markup they contain will always be interpreted and displayed in the same way whenever the page is visited by a user.

As you know from using the World Wide Web, modern websites can do much more; the pages we routinely visit are often far from static. They can contain "live" data, such as share prices or flight arrival times, animated displays with changing colors and fonts, or interactive capabilities such as the ability to click through a gallery of photographs or sort a column of data.

These clever capabilities are provided to the user by programs—often known as *scripts*—operating behind the scenes to manipulate what's displayed in the browser.

Note

The term *script* has no doubt been borrowed from the world of theater and TV, where the *script* defines the actions of the presenters or performers. In the case of a web page, the protagonists are the elements on the page, with a *script* provided by a scripting language such as, in this case, JavaScript. *Program* and *script* are, for our purposes, pretty much interchangeable terms, as are *programming* and *scripting*. You'll find all of these used in the course of the book.

Server- Versus Client-Side Programming

There are two fundamental ways of adding scripts to otherwise static web content:

- You can have the web server execute a script before delivering your page to the user. Such scripts can define what information is sent to the browser for display to the user—for example, by retrieving product prices from the database of an online store, checking a user's identity credentials before logging her into a private area of the website, or retrieving the contents of an email mailbox. These scripts are generally run at the web server *before* generating the requested web page and serving it to the user. You won't be surprised to learn that we refer to this as *server-side scripting*.
- Alternatively, the scripts themselves, rather than being run on the server, can be delivered to the user's browser along with the code of the page. Such scripts are then executed by the browser and operate on the page's already-delivered content. The many functions such scripts can perform include animating page sections, reformatting page layouts, allowing the user to drag-and-drop items within a page, validating user input on forms, redirecting users to other pages, and much more. You have probably already guessed that this is referred to as *client-side scripting*, and you're correct.

This book is all about JavaScript, the most-used language for client-side scripting on the Internet.

Note

There is, in fact, an elegant way to incorporate output from server-side scripts into your client-side JavaScript programs. We look at this in Part V, "Using JavaScript Libraries," when we study a technique called *Ajax*.

JavaScript in a Nutshell

Note

Although the names are similar, JavaScript doesn't have much, if anything, to do with the Java language developed by Sun Microsystems. The two languages share some aspects of syntax, but no more so than either of them do with a whole host of other programming languages.

A program written in JavaScript can access the elements of a web page, and the browser window in which it is running, and perform actions on those elements, as well as create new page elements. A few examples of JavaScript's capabilities include

- Opening new windows with a specified size, position, and style (for example, whether the window has borders, menus, toolbars, and so on)
- Providing user-friendly navigation aids such as drop-down menus
- Validation of data entered into a web form to make sure that it is of an acceptable format before the form is submitted to the web server
- Changing how page elements look and behave when particular events occur, such as the mouse cursor moving over them
- Detecting and exploiting advanced features supported by the particular browser being used, such as third-party plug-ins, or native support for new technologies

Because JavaScript code runs locally inside the user's browser, the page tends to respond quickly to JavaScript instructions, enhancing the user's experience and making the application seem more like one of the computer's native applications rather than simply a web page. Also, JavaScript can detect and utilize certain user actions that HTML can't, such as individual mouse clicks and keyboard actions.

Virtually every web browser in common use has support for JavaScript.

Where JavaScript Came From

The ancestry of JavaScript dates back to the mid 1990s, when version 1.0 was introduced for Netscape Navigator 2.

The European Computer Manufacturers Association (ECMA) became involved, defining ECMAScript, the great-granddaddy of the current language. At the same time, Microsoft introduced jScript, its own version of the language, for use in its Internet Explorer range of browsers.

Tip

ECMA continues to issue updated versions of the ECMAScript language standard. At the time of writing, ECMAScript 6 is nearing its final version, and in Part VI, “Advanced Topics,” you can read about some of the new language features soon to become available.

Note

JavaScript is not the only client-side scripting language. Microsoft browsers have supported (in addition to jScript, Microsoft’s version of JavaScript) a scripting-oriented version of the company’s own Visual Basic language, called VBScript.

JavaScript, however, has much better browser support; a version of JavaScript is supported by nearly every modern browser.

The Browser Wars

In the late 1990s, Netscape Navigator 4 and Internet Explorer 4 both claimed to offer major improvements over earlier browser versions in terms of what could be achieved with JavaScript.

Unfortunately, the two sets of developers had gone in separate directions, each defining its own additions to the JavaScript language, and how it interacted with your web page.

This ludicrous situation forced developers to essentially write two versions of each of their scripts, and use some clumsy and often error-prone routines to try to determine which browser was being used to view the page, and subsequently switch to the most appropriate version of their JavaScript code.

Note

The World Wide Web Consortium (W3C) is an international community that exists to develop open standards to support the long-term growth of the World Wide Web. Its website at http://www.w3.org/ is a vast resource of information and tools relating to web standards.

Thankfully, the World Wide Web Consortium (the W3C) worked hard with the individual browser manufacturers to standardize the way web pages were constructed and manipulated, by means of the Document Object Model (DOM). Level 1 of the new standardized DOM was completed in late 1998, and Level 2 in late 2000.

Don’t worry if you’re not sure what the DOM is or what it does—you learn a lot about it later this hour and through the course of this book.

The `<script>` Tag

Whenever the page is requested by a user, any JavaScript programs it contains are passed to the browser along with page content.

Note

JavaScript is an *interpreted* language, rather than a *compiled* language such as C++ or Java. The JavaScript instructions are passed to the browser as plain text and are interpreted sequentially; they do not need to first be “compiled” into condensed machine code only readable by the computer’s processor. This offers big advantages in that JavaScript programs are easy to read, can be edited swiftly, and their operation can be retested simply by reloading the web page in the browser.

You can include JavaScript statements directly into your HTML code by placing them between `<script>` and `</script>` tags within the HTML:

Click here to view code image

```
<script>
    ... JavaScript statements ...
</script>
```

The examples in this book are all written to validate correctly as HTML5, in which no obligatory attributes are specified for the `<script>` element (the `type` attribute is optional in HTML5, and has been excluded from the examples in this book to aid clarity). However, if you write JavaScript for inclusion in HTML 4.x or XHTML pages, you should add the `type` attribute to your `<script>` elements:

Click here to view code image

```
<script type="text/javascript">
    ... JavaScript statements ...
</script>
```

You’ll also occasionally see `<script>` elements having the attribute `language="JavaScript"`. This has long been deprecated, so unless you think you need to support ancient browsers such as Navigator and Mosaic, there’s no need to continue writing code like this.

Note

The term *deprecated* is applied to software features or practices to indicate that they are best avoided, usually because they have been superseded.

Although still supported to provide backward compatibility, their *deprecated* status often implies that such features will be removed in the near future.

The examples in this hour place their JavaScript code within the body section of the document, but JavaScript code can appear elsewhere in the document too; you can also use the `<script>` element to load JavaScript code saved in an external file. We discuss how to include JavaScript in your pages in much more detail in Hour 2, "Writing Simple Scripts."

Introducing the DOM

A Document Object Model (DOM) is a conceptual way of visualizing a document and its contents.

Each time your browser is asked to load and display a page, it needs to interpret (we usually use the word "parse") the source code contained in the HTML file comprising the page. As part of this parsing process, the browser creates a type of internal model known as a DOM representation based on the content of the loaded document. It is this model that the browser then refers to when rendering the visible page. You can use JavaScript to access and edit the various parts of the DOM representation, at the same time changing the way the user sees and interacts with the page in view.

In the early days, JavaScript provided rather primitive access to certain parts of a web page. JavaScript programs could gain access, for example, to the images and forms contained in a web page; a JavaScript program could contain statements to select "the second form on the page" or "the form called 'registration'."

Web developers sometimes refer to this as DOM Level 0, in backward-compatible homage to the W3C's subsequent Level 1 DOM definition. As well as DOM Level 0, you might also see reference to the BOM, or Browser Object Model. Since then, the W3C has gradually extended and improved the DOM specification. The W3C's much more ambitious definition has produced a DOM that is valid not just for web pages and JavaScript, but for any programming language and for XML, in addition to HTML, documents.

Note

In this book, we concentrate on the W3C's DOM Levels 1 and 2 DOM definitions. If you're interested in the details of the various DOM levels, you can find a good overview at https://developer.mozilla.org/en/DOM_Levels.

The W3C and Standards Compliance

The browser vendors have incorporated much-improved support for DOM in their most recent versions. At the time of writing, Internet Explorer is shipping in version 11, Netscape Navigator has been reborn as Mozilla Firefox (currently in version 35.0), and other competitors in the market include Opera, Konqueror, Apple's Safari, and

Google's Chrome and Chromium. All of these offer excellent support for the DOM.

The situation has improved markedly for web developers. Apart from a few irritating quirks, we can now largely forget about writing special code for individual browsers provided that we follow the DOM standards.

Note

The use of early browsers such as Netscape Navigator (any version) and Internet Explorer up to version 5.5 has now virtually disappeared. This book concentrates on more modern browsers that are compatible with DOM Level 1 or better, such as Internet Explorer 9+, Firefox, Google Chrome, Apple Safari, Opera, and Konqueror. You are recommended to upgrade your browser to the latest stable version.

The `window` and `document` Objects

Each time your browser loads and displays a page, it creates in memory an internal representation of the page and all its elements, the DOM. In the DOM, elements of your web page have a logical, hierarchical structure, like a tree of interconnected parent and child *objects*. These objects, and their interconnections, form a conceptual model of the web page and the browser that contains and displays it. Each object also has a list of *properties* that describe it, and a number of *methods* we can use to manipulate those properties using JavaScript.

Right at the top of the hierarchical tree is the browser `window` object. This object is a parent or ancestor to everything else in the DOM representation of your page.

The `window` object has various child objects, some of which are visualized in Figure 1.1. The first child object shown in Figure 1.1, and the one we'll use most in this book, is the `document` object. Any HTML page loaded into the browser creates a `document` object containing all of the HTML and other resources that go into making up the displayed page. All of this information is accessible via JavaScript as a parent-child hierarchy of objects, each with its own properties and methods.

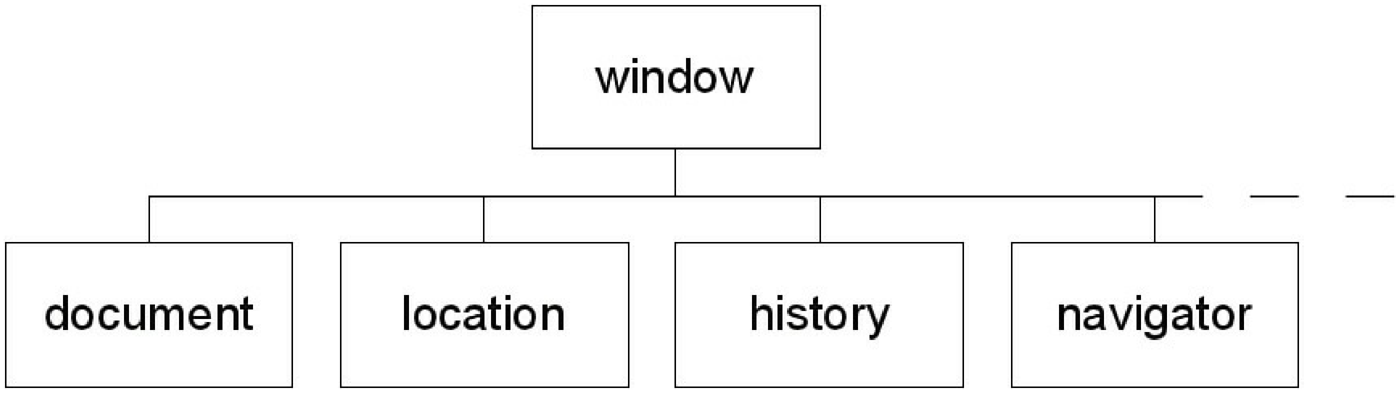


FIGURE 1.1 The `window` object and some of its children

The other children of the `window` object visible in Figure 1.1 are the `location` object (containing details of the URL of the currently loaded page), the `history` object (containing details of the browser's previously visited pages), and the `navigator` object (which stores details of the browser type, version, and capabilities). We look in detail at these objects in Hour 4, "DOM Objects and Built-In Objects," and use them again at intervals throughout the book, but for now let's concentrate on the `document` object.

Object Notation

The notation we use to represent objects within the tree uses the dot or period:

```
parent.child
```

As an example, referring to Figure 1.1, the `location` object is a child of the `window` object, so in the DOM it is referred to like this:

```
window.location
```

Tip

This notation can be extended as many times as necessary to represent any object in the tree. For example

```
object1.object2.object3
```

represents `object3`, whose parent is `object2`, which is itself a child of `object1`.

The `<body>` section of your HTML page is represented in the DOM as a child element of the `document` object; we would access it like this:

```
window.document.body
```

The last item in the sequence can also be, instead of another object, a *property* or *method* of the parent object:

```
    object1.object2.property
    object1.object2.method
```

For example, let's suppose that we want to access the `title` property of the current document, as specified by the HTML `<title>...</title>` tags. We can simply use

```
window.document.title
```

Note

Don't worry if object hierarchy and dot notation don't seem too clear right now. You'll be seeing many examples in the course of the book!

Tip

The `window` object always contains the current browser window, so you can refer to `window.document` to access the current document. As a shortcut, you can also simply use `document` instead of `window.document`—this also refers to the current document.

If you have several windows open, or if you are using a frameset, there will be a separate `window` and `document` object for each window or frame. To refer to one of these documents, you need to use the relevant window name and document name belonging to the window or frame in question.

Talking to the User

Let's take a look at some of the methods associated with the `window` and `document` objects. We begin with two methods, each of which provides a means to talk to the user.

window.alert()

Even if you don't realize it, you've seen the results of the `window` object's `alert` method on many occasions. The `window` object, you'll recall, is at the top of the DOM hierarchy, and represents the browser window that's displaying your page. When you call the `alert()` method, the browser pops open a dialog displaying your message, along with an OK button. Here's an example:

Click here to view code image

```
<script>window.alert("Here is my message");</script>
```

This is our first working example of the dot notation. Here we are calling the `alert()` method of the `window` object, so our `object.method` notation becomes `window.alert`.

Tip

In practice, you can leave out the `window.` part of the statement. Because the `window` object is the top of the DOM hierarchy (it's sometimes referred to as the *global object*), any methods called without direct reference to their parent object are assumed to belong to `window`. So

Click here to view code image

```
<script>alert("Here is my message");</script>
```

works just as well.

Notice that the line of text inside the parentheses is contained within quotation marks. These can be single or double quotes, but they must be there, or an error will be produced.

This line of code, when executed in the browser, pops up a dialog like the one in Figure 1.2.

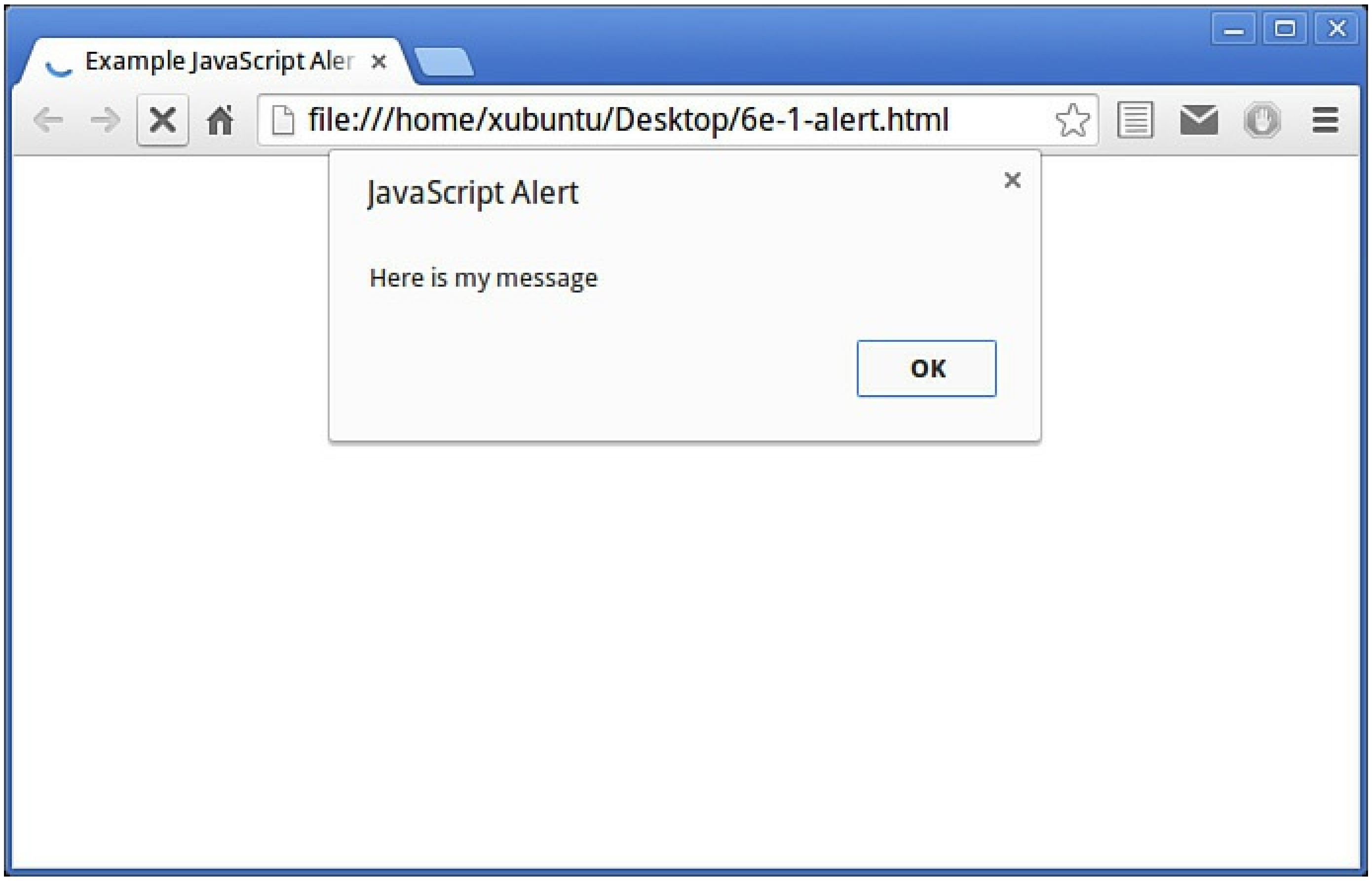


FIGURE 1.2 A `window.alert()` dialog

Tip

Figure 1.2 shows the alert generated by the Chrome browser running on Ubuntu Linux. The appearance of the dialog changes in detail depending on the particular browser, operating system, and display options you are using, but it always contains the message along with a single OK button.

Tip

Until the user clicks OK, he is prevented from doing anything else with the page.

A dialog that behaves this way is known as a *modal* dialog.

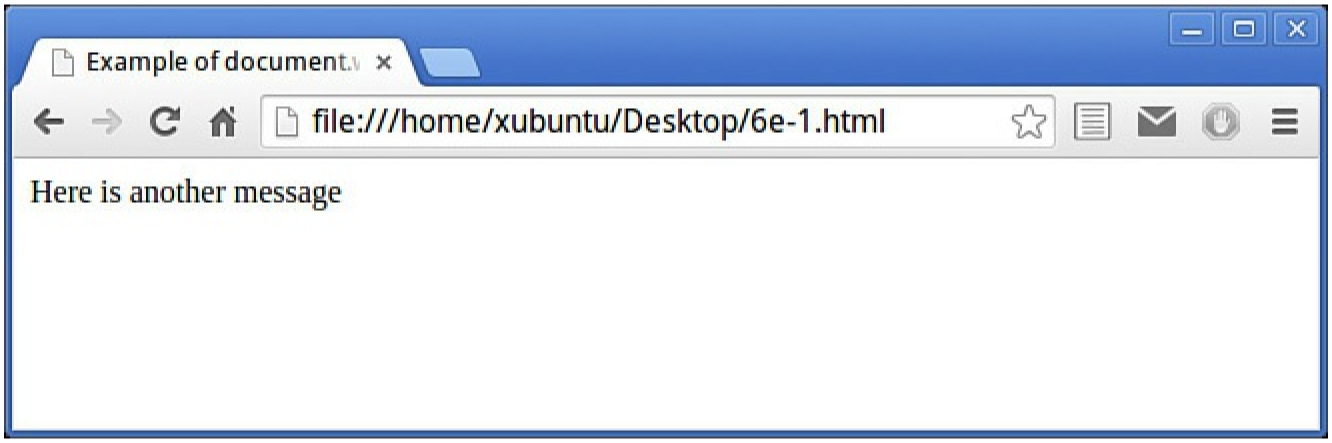
---

document.write()

You can probably guess what the `write` method of the document object does, simply from its name. This method, instead of popping up a dialog, writes characters directly into the DOM of the document, as shown in Figure 1.3.

Click here to view code image

```
<script>document.write("Here is another message");</script>
```



Here is another message

FIGURE 1.3 Using `document.write()`

Note

In fact, `document.write` is a pretty dumb way to write content to the page—it has a lot of limitations, both in terms of its function and in terms of coding style and maintainability. It has largely fallen into disuse for "serious" JavaScript programming. By the time you come to write more advanced JavaScript programs, you'll have learned much better ways to put content into your pages using JavaScript and the DOM. However, we use `document.write` quite a lot during Part I of the book, while you come to grips with some of the basic principles of the language.

Try it Yourself: "Hello World!" in JavaScript

It seems almost rude to introduce a programming language without presenting the traditional "Hello World" example. Have a look at the simple HTML document of Listing 1.1.

LISTING 1.1 "Hello World!" in an alert() Dialog

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello from JavaScript!</title>
</head>
<body>
    <script>
        alert("Hello World!");
    </script>
</body>
</html>
```

Create a document called hello.html in your text editor, and enter the preceding code. Save it to a convenient place on your computer, and then open it with your web browser.

Caution

Some text editor programs might try to add a .txt extension to the filename you specify. Be sure your saved file has the extension .html or the browser will probably not open it correctly.

Many popular operating systems allow you to right-click on the icon of the HTML file and choose Open With..., or similar wording. Alternatively, fire up your chosen browser, and use the **File > Open** options from the menu bar to navigate to your file and load it into the browser.

You should see a display similar to Figure 1.2, but with the message "Hello World!" in the dialog. If you have more than one browser installed on your computer, try them all, and compare the display—the dialogs will probably look a little different, but the message, and the operation of the OK button, should be just the same.

Caution

The default security settings in some browsers cause them to show a security warning when they are asked to open local content, such as a file on your own computer. If your browser does this, just choose the option that allows the content to be shown.

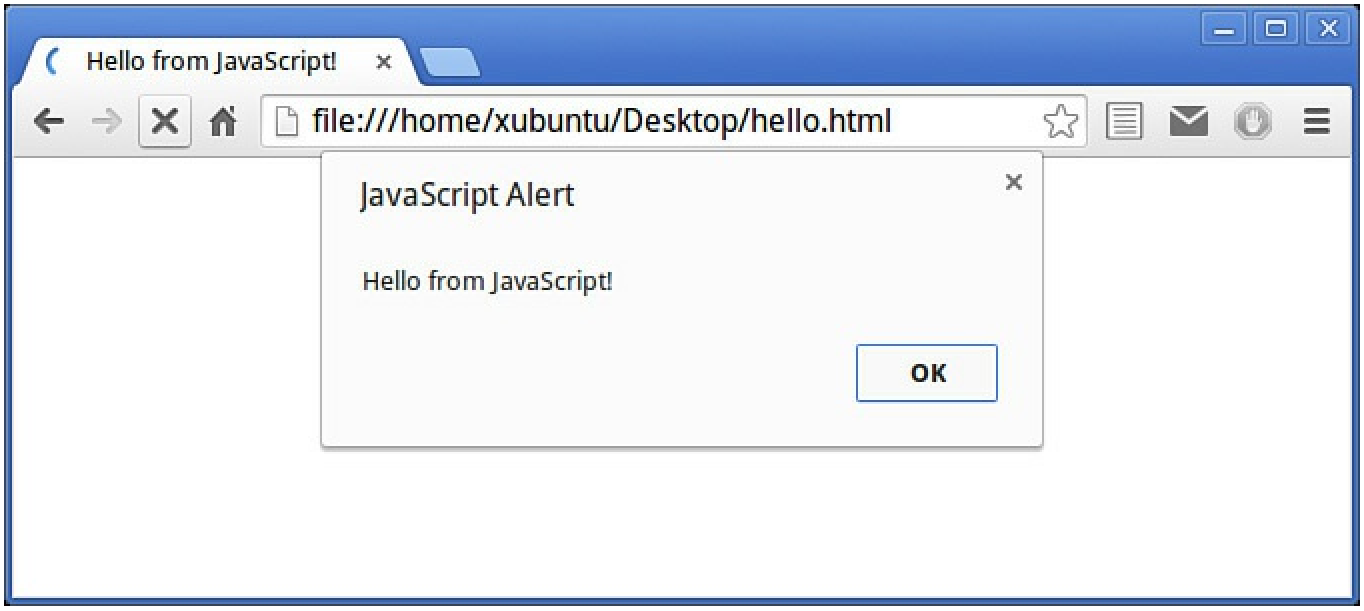
Reading a Property of the `document` Object

You may recall from earlier in the hour that objects in the DOM tree have properties and methods. You saw how to use the `write` method of the `document` object to output text to the page—now let's try *reading* one of the properties of the `document` object. We're going to use the `document.title` property, which contains the title as defined in the HTML `<title>` element of the page.

Edit hello.html in your text editor, and change the call to the `window.alert()` method:

```
alert(document.title);
```

Notice that `document.title` is NOT now enclosed in quotation marks—if it were, JavaScript would infer that we wanted to output the string "document.title" as literal text. Without the quote marks, JavaScript sends to the `alert()` method the *value* contained in the `document.title` property. The result is shown in Figure 1.4.



Hello from JavaScript!

FIGURE 1.4 Displaying a property of the `document` object

Summary

In this hour, you were introduced to the concepts of server-side and client-side scripting and had a brief history lesson about JavaScript and the Document Object Model. You had an overview of the sorts of things JavaScript can do to enhance your web pages and improve the experience for your users.

Additionally, you learned about the basic structure of the Document Object Model, and how JavaScript can access particular objects and their properties, and use the methods belonging to those objects.

In the lessons that follow, we'll build on these fundamental concepts to get into more advanced scripting projects.

Q&A

Q. If I use server-side scripting (in a language such as PHP or ASP), can I still use JavaScript on the client side?

A. Most definitely. In fact, the combination of server-side and client-side scripting provides a potent platform, capable of producing powerful applications. Google's Gmail is a good example.

Q. How many different browsers should I test in?

A. As many as you practically can. Writing standards-compliant code that avoids browser-specific features will go a long way toward making your code run smoothly in different browsers. However, one or two minor differences between browser implementations of certain features are likely to always exist.

Q. Won't the inclusion of JavaScript code slow down the load time of my pages?

A. Yes, though usually the difference is small enough not to be noticeable. If you have a particularly large piece of JavaScript code, you may feel it's worthwhile testing your page on the slowest connection a user is likely to have. Other than in extreme circumstances, it's unlikely to be a serious issue.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. Is JavaScript a compiled or an interpreted language?

a. A compiled language

b. An interpreted language

c. Neither

d. Both

2. What extra tags must be added to an HTML page to include JavaScript statements?

a. `<script>` and `</script>`

b. `<type="text/javascript">`

c. `<!--` and `-->`

3. The top level of the DOM hierarchy is occupied by:

a. The `document` property

b. The `document` method

c. The `document` object

d. The `window` object

Answers

1. b. JavaScript is an interpreted language. The program code is written in plain text, and the statements are read and executed one at a time.

2. a. JavaScript statements are added between `<script>` and `</script>` tags.

3. d. The `window` object is at the top of the DOM tree, and the `document` object is one of its child objects.

Exercises

- In the "Try It Yourself" section of this hour, we used the line

  ```
  alert(document.title);
  ```

  to output the `title` property of the `document` object. Try rewriting that script to instead output the `document.lastModified` property, which contains the date and time that the web page was last changed. (Be careful—property names are case sensitive. Note the capital M.) See whether you can then modify the code to use `document.write()` in place of `alert()` to write the property directly into the page, as in Figure 1.3.
- Try the example code from this hour in as many different browsers as you have access to. What differences do you note in how the example pages are displayed?

Hour 2. Writing Simple Scripts

What You’ll Learn in This Hour:

- Various ways to include JavaScript in your web pages
- The basic syntax of JavaScript statements
- How to declare and use variables
- Using mathematical operators
- How to comment your code
- Capturing mouse events

You learned in Hour 1, “Introducing JavaScript,” that JavaScript is a scripting language capable of making web pages more interactive.

In this hour you learn more about how JavaScript can be added to your web page, and then about some of the fundamental syntax of your JavaScript programs such as statements, variables, operators, and comments. You’ll also get your hands dirty with more code examples.

Including JavaScript in Your Web Page

In the previous hour I said that JavaScript programs are passed to the browser along with page content—but how do we achieve that? Actually there are two basic methods for associating JavaScript code with your HTML page, both of which use the `<script></script>` element introduced in Hour 1.

One method is to include the JavaScript statements directly into the HTML file, just like we did in the previous hour:

Click here to view code image

```
<script>
    ... Javascript statements are written here ...
</script>
```

A second, and usually preferable way to include your code is to save your JavaScript into a separate file, and use the `<script>` element to include that file by name using the `src` (source) attribute:

Click here to view code image

```
<script src='mycode.js'></script>
```

The preceding example includes the file `mycode.js`, which contains our JavaScript statements. If your JavaScript file is not in the same folder as the calling script, you can also add a (relative or absolute) path to it:

Click here to view code image

```
<script src='/path/to/mycode.js'></script>
```

or

Click here to view code image

```
<script src='http://www.example.com/path/to/mycode.js'></script>
```

Placing your JavaScript code in a separate file offers some important advantages:

- When the JavaScript code is updated, the updates are immediately available to any page using that same JavaScript file. This is particularly important in the context of JavaScript libraries, which we look at later in the book.
- The code for the HTML page is kept cleaner, and therefore easier to read and maintain.
- Performance is slightly improved because your browser caches the included file; therefore, having a local copy in memory next time it is needed by this or another page.

Note

It is customary to give files of JavaScript code the file extension .js, as in this example. However, your included code files can have any extension, and the browser will try to interpret the contents as JavaScript.

Caution

The JavaScript statements in the external file must NOT be surrounded by `<script> ... </script>` tags, nor can you place any HTML markup within the external file. Just include the raw JavaScript code.

Listing 2.1 shows the simple web page we used in Hour 1, but now with a file of JavaScript code included in the `<body>` section. JavaScript can be placed in either the head or body of the HTML page. In fact, it is more common—and generally recommended—for JavaScript code to be placed in the head of the page, where it provides a number of *functions* that can be called from elsewhere in the document. You learn about functions in Hour 3, “Using Functions”; until then, we limit ourselves to adding our example code into the body of the document.

LISTING 2.1 An HTML Document with a JavaScript File Included

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>A Simple Page</title>
</head>
<body>
    <p>Some content ...</p>
    <script src='mycode.js'></script>
</body>
</html>
```

When JavaScript code is added into the body of the document, the code statements are interpreted and executed as they are encountered while the page is being rendered. After the code has been read and executed, page rendering continues until the page is complete.

Tip

You're not limited to using a single `script` element—you can have as many of them on your page as you need.

Note

You sometimes see HTML-style comment notation `<!--` and `-->` inside `script` elements, surrounding the JavaScript statements, like this:

Click here to view code image

```
<script>
    <!--
    ... Javascript statements are written here ...
    -->
</script>
```

This was for the benefit of ancient browsers that didn't recognize the `<script>` tag. This HTML "comment" syntax prevented such browsers from displaying the JavaScript source code on the screen along with the page content. Unless you have a reason to support very old browsers, this technique is no longer required.

JavaScript Statements

JavaScript programs are lists of individual instructions that we refer to as *statements*. To interpret statements correctly, the browser expects to find each statement written on a separate line:

```
this is statement 1
this is statement 2
```

Alternatively, they can be combined in the same line by terminating each with a semicolon:

Click here to view code image

```
this is statement 1; this is statement 2;
```

To ease the readability of your code, and to help prevent hard-to-find syntax errors, it's good practice to combine both methods by giving each statement its own line and terminating the statement with a semicolon:

```
this is statement 1;
this is statement 2;
```

Commenting Your Code

Some statements are not intended to be executed by the browser's JavaScript interpreter, but are there for the benefit of anybody who may be reading the code. We refer to such lines as *comments*, and there are specific rules for adding comments to your code.

A comment that occupies just a single line of code can be written by placing a double forward slash before the content of the line:

```
// This is a comment
```

Note

JavaScript can also use the HTML comment syntax for single-line comments:

```
<!-- this is a comment -->
```

However, this is not commonly used in JavaScript programs.

To add a multiline comment in this way, we need to prefix every line of the comment:

```
// This is a comment
// spanning multiple lines
```

A more convenient way of entering multiline comments to your code is to prefix your comment with `/*` and terminate it with `*/`. A comment written using this syntax can span multiple lines:

```
/*  This comment can span
    multiple lines
    without needing
    to mark up every line  */
```

Adding comments to your code is really a good thing to do, especially when you're writing larger or more complex JavaScript applications. Comments can act as reminders to you, and also as instructions and explanations to anybody else reading your code at a later date.

Note

It’s true that comments add a little to the size of your JavaScript source file, and this can have an adverse effect on page-loading times. Generally the difference is so small as to be barely noticeable, but if it really matters you can always strip out all the comments from a “production” version of your JavaScript file—that is, a version to use with live, rather than development, websites.

Variables

A variable can be thought of as a named “pigeon-hole” where we keep a particular piece of data. Such data can take many different forms—an integer or decimal number, a string of characters, or various other data types discussed later in this hour or in those that follow. Our variables can be called pretty much anything we want, so long as we only use alphanumeric characters, the dollar sign $, or underscores in the name.

Note

JavaScript is case sensitive—a variable called `mypetcat` is a different variable from `Mypetcat` or `MYPETCAT`.

Many coders of JavaScript, and other programming languages, like to use the so-called *CamelCase* convention (also called mixedCase, BumpyCaps, and various other names) for variable names. In CamelCase, compound words or phrases have the elements joined without spaces, with each element’s initial letter capitalized except the first letter, which can be either upper- or lowercase. In this example, the variable would be named `MyPetCat` or `myPetCat`.

Let’s suppose we have a variable called `netPrice`. We can set the value stored in `netPrice` with a simple statement:

```
netPrice = 8.99;
```

We call this *assigning a value* to the variable. Note that we don’t have to declare the existence of this variable before assigning a value, as we would have to in some other programming languages. However, doing so is possible in JavaScript by using the `var` keyword, and in most cases is good programming practice:

```
var netPrice;
netPrice = 8.99;
```

Alternatively we can combine these two statements conveniently and readably into one:

```
var netPrice = 8.99;
```

To assign a *character string* as the value of a variable, we need to include the string in single or double quotes:

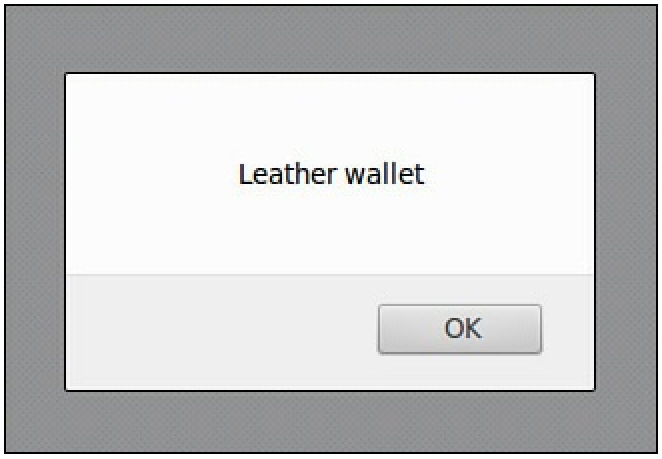
Click here to view code image

```
var productName = "Leather wallet";
```

We could then, for example, write a line of code sending the value contained in that variable to the `window.alert` method:

```
alert(productName);
```

The generated dialog would evaluate the variable and display it (this time, in Mozilla Firefox) as in Figure 2.1.



Leather wallet

FIGURE 2.1 Displaying the value of variable `productName`

Tip

Choose readable variable names. Having variable names such as `productName` and `netPrice` makes code much easier to read and maintain than if the same variables were called `var123` and `myothervar49`, even though the latter names are entirely valid.

Operators

The values we have stored in our variables aren't going to be much use to us unless we can manipulate them in calculations.

Arithmetic Operations

First, JavaScript allows us to carry out operations using the standard arithmetic operators of addition, subtraction, multiplication, and division.

```
var theSum = 4 + 3;
```

As you’ll have guessed, after this statement has been executed the variable `theSum` will contain a value of 7. We can use variable names in our operations too:

Click here to view code image

```
    var productCount = 2;
    var subtotal = 14.98;
    var shipping = 2.75;
    var total = subtotal + shipping;
```

We can use JavaScript to subtract (-), multiply (*), and divide (/) in a similar manner:

Click here to view code image

```
    subtotal = total - shipping;
    var salesTax = total * 0.15;
    var productPrice = subtotal / productCount;
```

To calculate the remainder from a division, we can use JavaScript’s *modulus division* operator. This is denoted by the % character:

Click here to view code image

```
    var itemsPerBox = 12;
    var itemsToBeBoxed = 40;
    var itemsInLastBox = itemsToBeBoxed % itemsPerBox;
```

In this example, the variable `itemsInLastBox` would contain the number *4* after the last statement completes.

JavaScript also has convenient operators to increment (++) or decrement (--) the value of a variable:

```
productCount++;
```

is equivalent to the statement

Click here to view code image

```
productCount = productCount + 1;
```

Similarly,

```
items--;
```

is just the same as

```
items = items - 1;
```

Tip

If you need to increment or decrement a variable by a value other than one, JavaScript also allows you to combine other arithmetic operators with the = operator; for example, += and -=.

The following two lines of code are equivalent:

```
total = total + 5;
total += 5;
```

So are these two:

```
counter = counter - step;
counter -= step;
```

We can use this notation for other arithmetic operators, such as multiplication and division:

```
price = price * uplift;
price *= uplift;
```

A more comprehensive list of JavaScript's arithmetic operators appears in Appendix B, "JavaScript Quick Reference."

Operator Precedence

When you use several operators in the same calculation, JavaScript uses *precedence rules* to determine in what order the calculation should be done. For example, consider the statement

```
var average = a + b + c / 3;
```

If, as the variable's name implies, you're trying to calculate an average, this would not give the desired result; the division operation would be carried out on `c` before adding the values of `a` and `b` to the result. To calculate the average correctly, we would have to add parentheses to our statement, like this:

Click here to view code image

```
var average = (a + b + c) / 3;
```

If you have doubts about the precedence rules, I would recommend that you always use parentheses liberally. There is no cost to doing so, it makes your code easier to read (both for you and for anyone else who later has to edit or decipher it), and it ensures that precedence issues don't spoil your calculations.

Note

If you have programming experience in another language such as PHP or Java, you'll find that the precedence rules in JavaScript are pretty much identical to the ones you're used to. You can find detailed information on JavaScript precedence at http://msdn.microsoft.com/en-us/library/z3ks45k7(v=vs.94).aspx.

Using the + Operator with Strings

Arithmetic operators don't make much sense if the variables they operate on contain strings rather than numeric values. The exception is the + operator, which JavaScript interprets as an instruction to concatenate (join together sequentially) two or more strings:

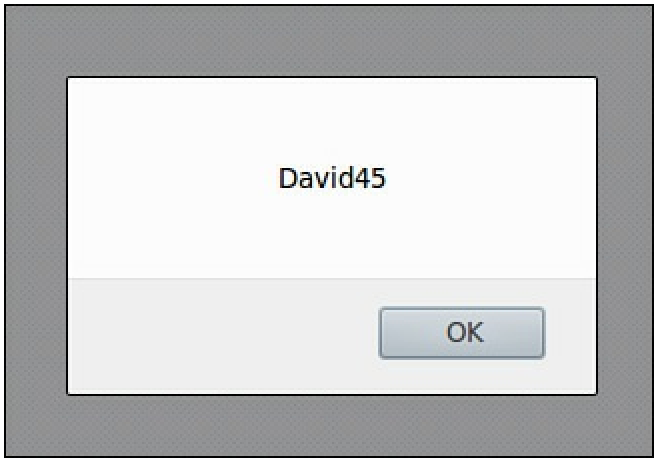
[Click here to view code image](#)

```
var firstname = "John";
var surname = "Doe";
var fullname = firstname + " " + surname;
// the variable fullname now contains the value "John Doe"
```

If you try to use the + operator on two variables, one of which is a string and the other numeric, JavaScript converts the numeric value to a string and concatenates the two:

```
var name = "David";
var age = 45;
alert(name + age);
```

[Figure 2.2](#) shows the result of using the + operator on a string and a numeric value.



```
David45
```

FIGURE 2.2 Concatenating a string and a numeric value

We talk about JavaScript data types, and string operations in general, much more in [Hour 5](#), "[Numbers and Strings](#)."

Try it Yourself: Convert Celsius to Fahrenheit

To convert a temperature in degrees Celsius to one measured in degrees Fahrenheit, we need to multiply by 9, divide by 5, and then add 32. Let's do that in JavaScript:

[Click here to view code image](#)

```
var cTemp = 100;  // temperature in Celsius
// Let's be generous with parentheses
var hTemp = ((cTemp * 9) /5 ) + 32;
```

In fact, we could have omitted all of the parentheses from this calculation and it would still have worked fine:

```
    var hTemp = cTemp*9/5 + 32;
```

However, the parentheses make the code easier to understand, and help prevent errors caused by operator precedence.

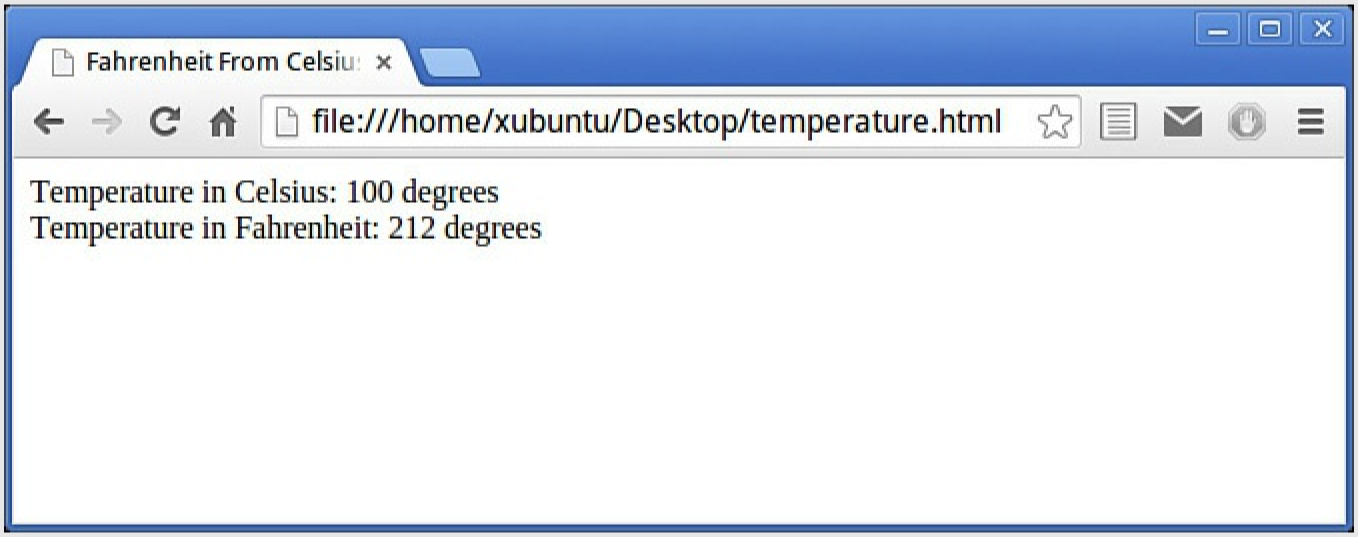
Let's test the code in a web page, as shown in Listing 2.2.

LISTING 2.2 Calculating Fahrenheit from Celsius

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Fahrenheit From Celsius</title>
</head>
<body>
    <script>
        var cTemp = 100;  // temperature in Celsius
        // Let's be generous with parentheses
        var hTemp = ((cTemp * 9) /5 ) + 32;
        document.write("Temperature in Celsius: " + cTemp + "
degrees<br/>");
        document.write("Temperature in Fahrenheit: " + hTemp + "
degrees");
    </script>
</body>
</html>
```

Save this code as a file `temperature.html` and load it into your browser. You should get the result shown in Figure 2.3.



```
Temperature in Celsius: 100 degrees
Temperature in Fahrenheit: 212 degrees
```

FIGURE 2.3 The output of Listing 2.2

Edit the file a few times to use different values for `cTemp`, and check that everything works OK.

Capturing Mouse Events

One of the fundamental purposes of JavaScript is to help make your web pages more interactive for the user. To achieve this, we need to have some mechanisms to detect what the user and the program are doing at any given moment—where the mouse is in the browser window, whether the user has clicked a mouse button or pressed a keyboard key, whether a page has fully loaded in the browser, and so on.

All of these occurrences we refer to as *events*, and JavaScript has a variety of tools to help us work with them. Let's take a look at some of the ways we can detect a user's mouse actions using JavaScript.

JavaScript deals with events by using so-called *event handlers*. We are going to investigate three of these: `onClick`, `onMouseOver`, and `onMouseOut`.

The `onClick` Event Handler

The `onClick` event handler can be applied to nearly all HTML elements visible on a page. One way we can implement it is to add one more attribute to the HTML element:

Click here to view code image

```
onclick=" ...some JavaScript code... "
```

Note

While adding event handlers directly into HTML elements is perfectly valid, it's not regarded these days as good programming practice. It serves us well for the examples in Part I of this book, but later in the book you learn more powerful and elegant ways to use event handlers.

Let's see an example; have a look at Listing 2.3.

LISTING 2.3 Using the `onClick` Event Handler

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>onClick Demo</title>
</head>
<body>
    <input type="button" onclick="alert('You clicked the button!')"
value="Click Me" />
</body>
```

```
</html>
```

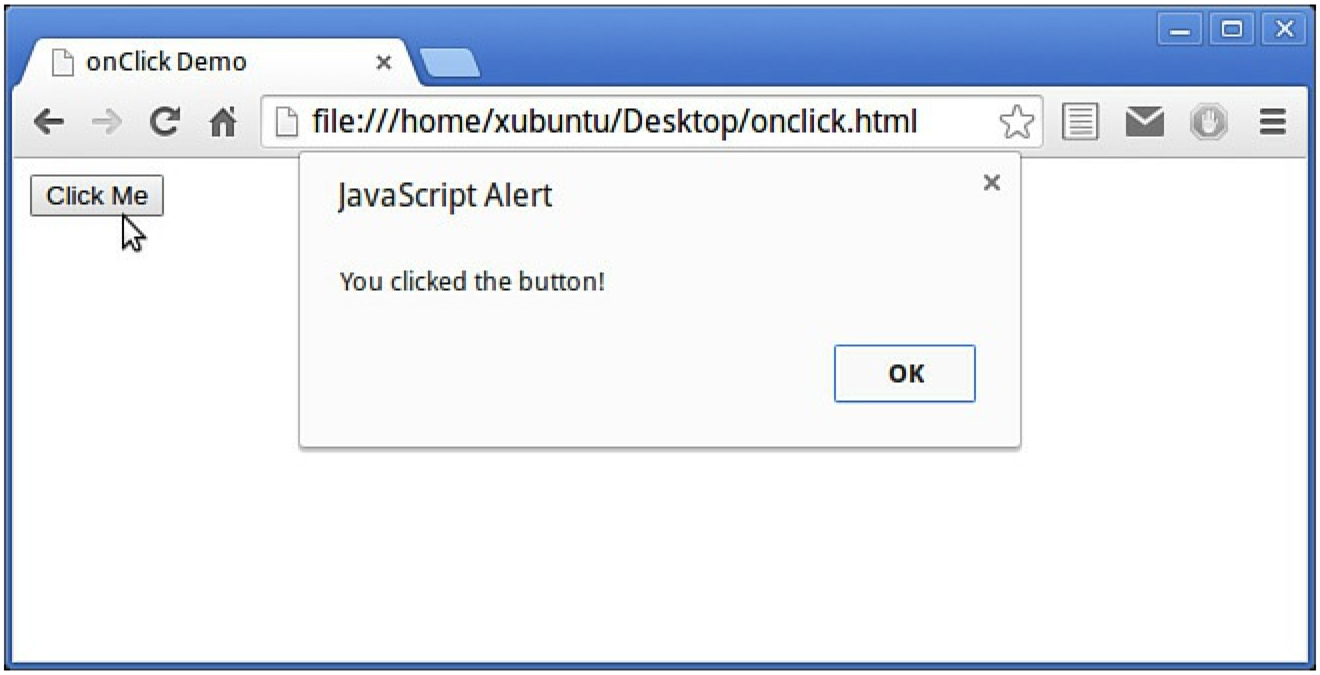
The HTML code adds a button to the `<body>` element of the page, and supplies that button with an `onclick` attribute. The value given to the `onclick` attribute is the JavaScript code we want to run when the HTML element (in this case a button) is clicked. When the user clicks on the button, the `onclick` *event* is activated (we normally say the event has been "fired") and the JavaScript statement(s) listed in the value of the attribute are executed.

In this case, there's just one statement:

Click here to view code image

```
alert('You clicked the button!')
```

Figure 2.4 shows the result of clicking the button.



You clicked the button!

FIGURE 2.4 Using the `onClick` event handler

Note

You may have noticed that we call the handler `onClick`, but that we write this in lowercase as `onclick` when adding it to an HTML element. This convention has arisen because, although HTML is case insensitive, XHTML is case *sensitive* and requires all HTML elements and attribute names to be written in lowercase.

onMouseOver and onMouseOut Event Handlers

When we simply want to detect where the mouse pointer is on the screen with reference to a particular page element, `onMouseOver` and `onMouseOut` can help us to do that.

The onMouseOver event is fired when the user's mouse cursor enters the region of the screen occupied by the element in question. The onMouseOut event, as I'm sure you've already guessed, is fired when the cursor leaves that same region.

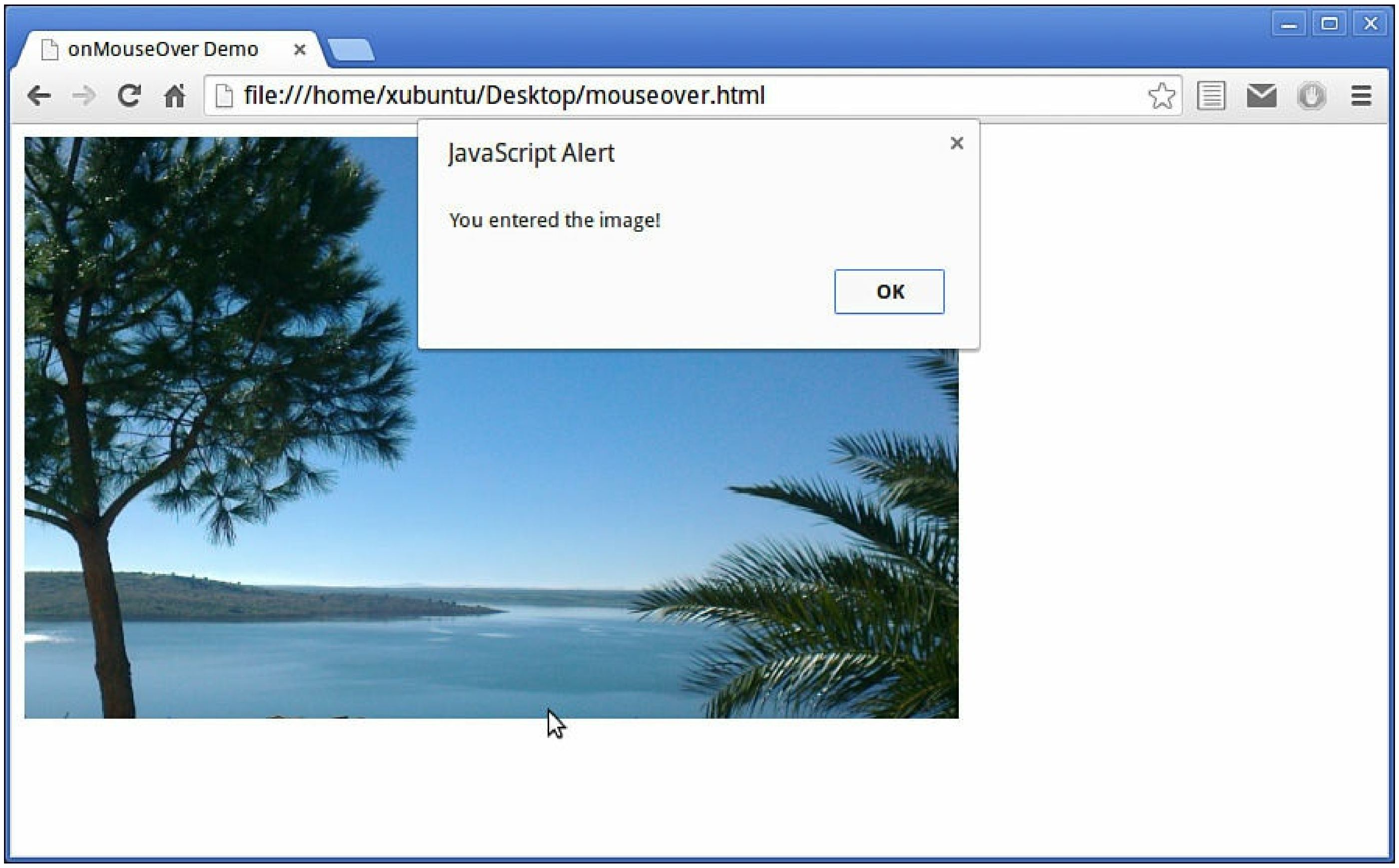
Listing 2.4 provides a simple example of the onMouseOver event in action.

LISTING 2.4 Using **onMouseOver**

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>onMouseOver Demo</title>
</head>
<body>
    <img src="image1.png" alt="image 1" onmouseover="alert('You entered
the image!')" />
</body>
</html>
```

The result of running the script is shown in Figure 2.5. Replacing onmouseover with onmouseout in the code will, of course, simply fire the event handler—and therefore pop up the alert dialog—as the mouse *leaves* the image, rather than doing so as it enters.



You entered the image!

FIGURE 2.5 Using the `onMouseOver` event handler

Try it Yourself: Creating an Image Rollover

We can use the `onMouseOver` and `onMouseOut` events to change how an image appears while the mouse pointer is above it. To achieve this, we use `onMouseOver` to change the `src` attribute of the HTML `<img>` element as the mouse cursor enters, and `onMouseOut` to change it back as the mouse cursor leaves. The code is shown in Listing 2.5.

LISTING 2.5 An Image Rollover Using onMouseOver and onMouseOut

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>OnMouseOver Demo</title>
</head>
<body>
    <img src="tick.gif" alt="tick" onmouseover="this.src='tick2.gif';"
onmouseout="this.src='tick.gif';" />
</body>
</html>
```

You may notice something new in the syntax we used here. Within the JavaScript statements for `onMouseOver` and `onMouseOut` we use the keyword `this`.

When using `this` within an event handler added via an attribute of an HTML element, `this` refers to the HTML element itself; in this case, you can read it as “this image,” and `this.src` refers (using the “dot” notation that we’ve already met) to the `src` (source) property of this image object.

In this example we used two images, `tick.gif` and `tick2.gif`—you can use any images you have on hand, but the demonstration works best if they are the same size, and not too large.

Use your editor to create an HTML file containing the code of Listing 2.5. You can change the image names tick.gif and tick2.gif to the names of your two images, if different; just make sure the images are saved in the same folder as your HTML file. Save the HTML file and open it in your browser.

You should see that the image changes as the mouse pointer enters, and changes back as it leaves, as depicted in Figure 2.6.

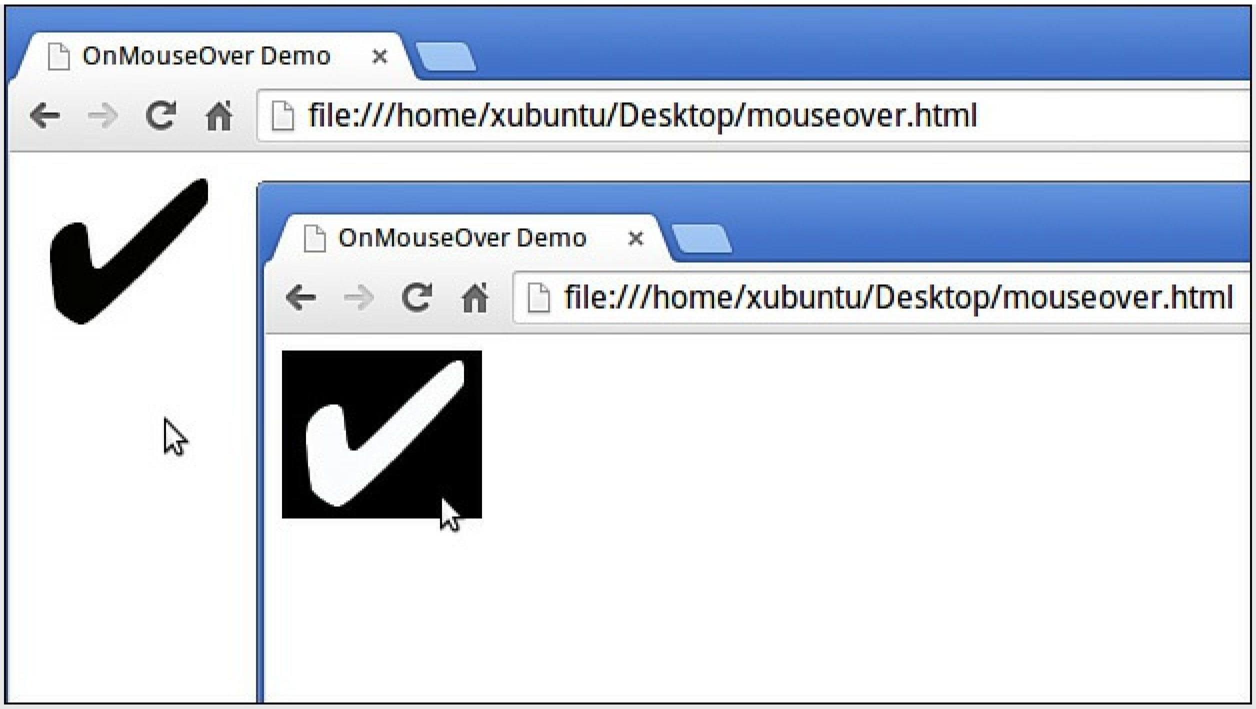

OnMouseOver Demo
file:///home/xubuntu/Desktop/mouseover.html
OnMouseOver Demo
file:///home/xubuntu/Desktop/mouseover.html

FIGURE 2.6 An image rollover using `onMouseOver` and `onMouseOut`

Note

There was a time when image rollovers were regularly done this way, but these days they can be achieved much more efficiently using Cascading Style Sheets (CSS). Still, it's a convenient way to demonstrate the use of the `onMouseOver` and `onMouseOut` event handlers.

Summary

You covered quite a lot of ground this hour.

First of all you learned various ways to include JavaScript code in your HTML pages.

You studied how to declare variables in JavaScript, assign values to those variables, and manipulate them using arithmetic operators.

Finally, you were introduced to some of JavaScript's event handlers, and you learned how to detect certain actions of the user's mouse.

Q&A

Q. Some of the listings and code snippets list opening and closing `<script>` tags on the same line; other times they are on separate lines. Does it matter?

A. Empty spaces, such as the space character, tabs, and blank lines, are completely ignored by JavaScript. You can use such blank space, which programmers usually call *whitespace*, to lay out your code in such a way that it's more legible and easy to follow.

Q. Can I use the same `<script>` element both to include an external JavaScript file and to contain JavaScript statements?

A. No. If you use the `script` element to include an external JavaScript file by using the `src` attribute, you cannot also include JavaScript statements between `<script>` and `</script>`—this region must be left empty.

Workshop

Try to answer the following questions before looking at the "Answers" section that follows.

Quiz

1. What is an `onClick` event handler?

   a. An object that detects the mouse's location in the browser

   b. A script that executes in response to the user clicking the mouse

   c. An HTML element that the user can click

2. How many `<script>` elements are permitted on a page?

   a. None

   b. Exactly one

   c. Any number

3. Which of these is NOT a true statement about variables?

   a. Their names are case sensitive.

   b. They can contain numeric or non-numeric information.

   c. Their names may contain spaces.

Answers

1. b. An `onClick` event handler is a script that executes when the user clicks the mouse.
2. c. You can use as many `<script>` elements as you need.
3. c. Variable names in JavaScript must not contain spaces.

Exercises

- Starting with Listing 2.4, remove the `onMouseOver` and `onMouseOut` handlers from the `<img>` element. Instead, add an `onClick` handler to set the `title` property of the image to `My New Title`. (Hint: You can access the image title using `this.title`.)

- Can you think of an easy way to test whether your script has correctly set the new image title?

Hour 3. Using Functions

What You'll Learn in This Hour:

- How to define functions
- How to call (execute) functions
- How functions receive data
- Returning values from functions
- About the scope of variables

Commonly, programs carry out the same or similar tasks repeatedly during the course of their execution. For you to avoid rewriting the same piece of code over and over again, JavaScript has the means to parcel up parts of your code into reusable modules, called *functions*. Once you've written a function, it is available for the rest of your program to use, as if it were itself a part of the JavaScript language.

Using functions also makes your code easier to debug and maintain. Suppose you've written an application to calculate shipping costs; when the tax rates or haulage prices change, you'll need to make changes to your script. There may be 50 places in your code where such calculations are carried out. When you attempt to change every calculation, you're likely to miss some instances or introduce errors. However, if all such calculations are wrapped up in a few functions used throughout the application, then you just need to make changes to those functions. Your changes will automatically be applied all through the application.

Functions are one of the basic building blocks of JavaScript and will appear in virtually every script you write. In this hour you see how to create and use functions.

General Syntax

Creating a function is similar to creating a new JavaScript command that you can use in your script.

Here's the basic syntax for creating a function:

Click here to view code image

```
function sayHello() {
    alert("Hello");
    // ... more statements can go here ...
}
```

You begin with the keyword `function`, followed by your chosen function name with parentheses appended, then a pair of curly braces, {}. Inside the braces go the

JavaScript statements that make up the function. In the case of the preceding example, we simply have one line of code to pop up an `alert` dialog, but you can add as many lines of code as are necessary to make the function...well, function!

Caution

The keyword `function` must always be used in lowercase, or an error will be generated.

To keep things tidy, you can collect together as many functions as you like into one `<script>` element:

```
<script>
    function doThis() {
        alert("Doing This");
    }
    function doThat() {
        alert("Doing That");
    }
</script>
```

Calling Functions

Code wrapped up in a function definition will not be executed when the page loads. Instead, it waits quietly until the function is *called*.

To call a function, you simply use the function name (with the parentheses) wherever you want to execute the statements contained in the function:

```
sayHello();
```

For example, you may want to add a call to your new function `sayHello()` to the `onClick` event of a button:

Click here to view code image

```
<input type="button" value="Say Hello" onclick="sayHello()" />
```

Tip

Function names, like variable names, are case-sensitive. A function called `MyFunc()` is different from another called `myFunc()`. Also, as with variable names, it’s really helpful to the readability of your code to choose meaningful function names.

Tip

You’ve already seen numerous examples of using the *methods* associated with

JavaScript objects, such as `document.write()` and `window.alert()`. Methods are simply functions that "belong" to a specific object. You learn much more about objects in Hour 4, "DOM Objects and Built-in Objects."

Putting JavaScript Code in the Page `<head>`

Up to now, our examples have all placed the JavaScript code into the `<body>` part of the HTML page. Using functions lets you employ the much more common, and usually preferable, practice of storing your JavaScript code in the `<head>` of the page. Functions contained within a `<script>` element in the page head, or in an external file included via the `src` attribute of a `<script>` element in the page head, are available to be called from anywhere on the page. Putting functions in the document's head section ensures that they have been defined prior to any attempt being made to execute them.

Listing 3.1 shows an example.

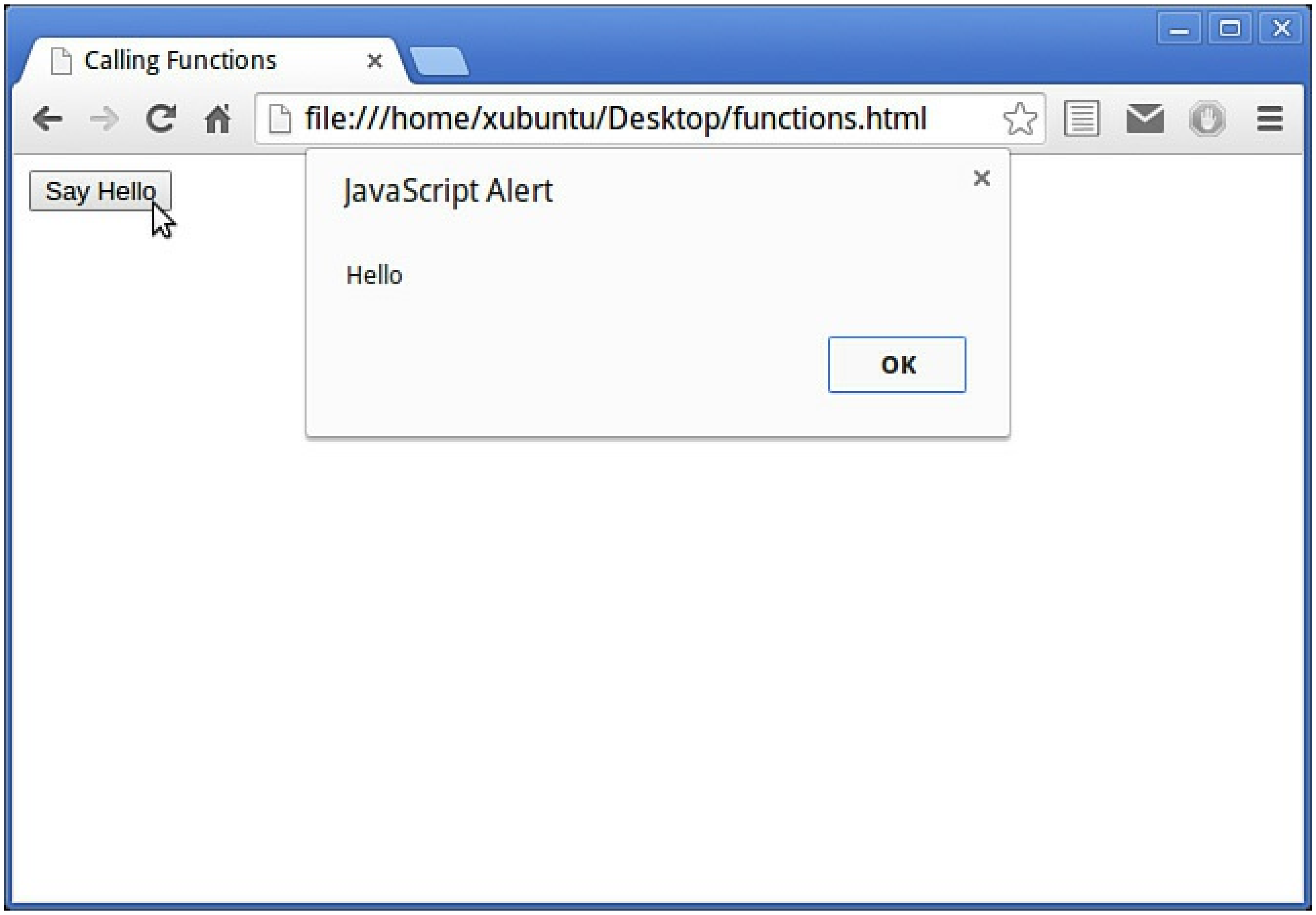
LISTING 3.1 Functions in the Page Head

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Calling Functions</title>
    <script>
        function sayHello() {
            alert("Hello");
        }
    </script>
</head>
<body>
    <input type="button" value="Say Hello" onclick="sayHello()" />
</body>
</html>
```

In this listing, you can see that the function definition itself has been placed inside a `<script>` element in the page head, but the call to the function has been made from a different place entirely—on this occasion, from the `onClick` event handler of a button in the body section of the page.

The result of clicking the button is shown in Figure 3.1.



Hello

FIGURE 3.1 Calling a JavaScript function

Passing Arguments to Functions

It would be rather limiting if your functions could only behave in an identical fashion each and every time they were called, as would be the case in the preceding example.

Fortunately, you can extend the capabilities of functions a great deal by passing data to them. You do this when the function is called, by passing to it one or more *arguments*:

```
functionName(arguments)
```

Let's write a simple function to calculate the cube of a number and display the result:

```
    function cube(x) {
        alert(x * x * x);
    }
```

Now we can call our function, replacing the variable *x* with a number. Calling the function like the following results in a dialog box being displayed that contains the result of the calculation, in this case 27:

```
cube(3);
```

Of course, you could equally pass a variable name as an argument. The following code would also generate a dialog containing the number 27:

```
var length = 3;
cube(length);
```

Note

You'll sometimes see or hear the word *parameters* used in place of *arguments*, but it means exactly the same thing.

Multiple Arguments

Functions are not limited to a single argument. When you want to send multiple arguments to a function, all you need to do is separate them with commas:

```
function times(a, b) {
    alert(a * b);
}
times(3, 4); // alerts '12'
```

You can use as many arguments as you want.

Caution

Make sure that your function calls contain enough argument values to match the arguments specified in the function definition. If any of the arguments in the definition are left without a value, JavaScript may issue an error, or the function may perform incorrectly. If your function call is issued with too many arguments, the extra ones will be ignored by JavaScript.

It's important to note that the names given to arguments in the definition of your function have nothing to do with the names of any variables whose values are passed to the function. The variable names in the argument list act like placeholders for the actual values that will be passed when the function is called. The names that you give to arguments are only used inside the function definition to specify how it works.

We talk about this in more detail later in the hour when we discuss variable *scope*.

Try it Yourself: A Function to Output User Messages

Let's use what we've learned so far in this hour by creating a function that can send the user a message about a button he or she has just clicked. We place the function definition in the `<head>` section of the page and call it with multiple arguments.

Here's our function:

Click here to view code image

```
function buttonReport(buttonId, buttonName, buttonValue) {
```

```
            // information about the id of the button
            var userMessage1 = "Button id: " + buttonId + "\n";
            // then about the button name
            var userMessage2 = "Button name: " + buttonName + "\n";
            // and the button value
            var userMessage3 = "Button value: " + buttonValue;
            // alert the user
            alert(userMessage1 + userMessage2 + userMessage3);
        }
```

The function `buttonReport` takes three arguments, those being the `id`, `name`, and `value` of the button element that has been clicked. With each of these three pieces of information, a short message is constructed. These three messages are then concatenated into a single string, which is passed to the `alert()` method to pop open a dialog containing the information.

Tip

You may have noticed that the first two message strings have the element `"\n"` appended to the string; this is a “new line” character, forcing the message within the alert dialog to return to the left and begin a new line. Certain special characters like this one must be prefixed with \ if they are to be correctly interpreted when they appear in a string. Such a prefixed character is known as an *escape sequence*. You learn more about escape sequences in Hour 5, “Numbers and Strings.”

To call our function, we put a button element on our HTML page, with its `id`, `name`, and `value` defined:

Click here to view code image

```
    <input type="button" id="id1" name="Button 1" value="Something" />
```

We need to add an `onClick` event handler to this button from which to call our function. We’re going to use the `this` keyword, as discussed in Hour 2, “Writing Simple Scripts”:

Click here to view code image

```
    onclick = "buttonReport(this.id, this.name, this.value)"
```

The complete listing is shown in Listing 3.2.

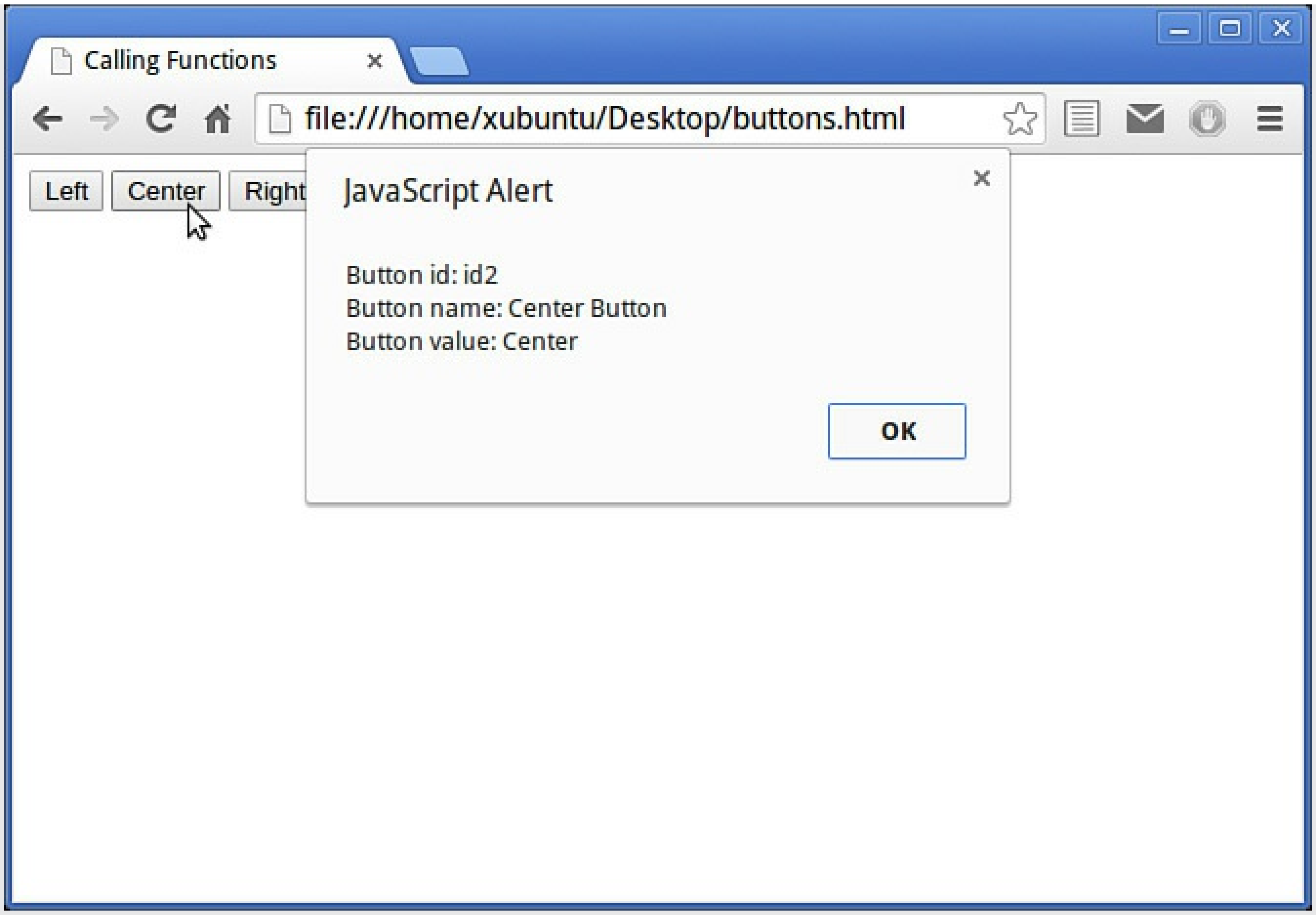
LISTING 3.2 Calling a Function with Multiple Arguments

Click here to view code image

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

```
<head>
    <title>Calling Functions</title>
    <script>
        function buttonReport(buttonId, buttonName, buttonValue) {
            // information about the id of the button
            var userMessage1 = "Button id: " + buttonId + "\n";
            // then about the button name
            var userMessage2 = "Button name: " + buttonName + "\n";
            // and the button value
            var userMessage3 = "Button value: " + buttonValue;
            // alert the user
            alert(userMessage1 + userMessage2 + userMessage3);
        }
    </script>
</head>
<body>
    <input type="button" id="id1" name="Left Hand Button" value="Left"
onclick ="buttonReport(this.id, this.name, this.value)"/>
    <input type="button" id="id2" name="Center Button" value="Center"
onclick ="buttonReport(this.id, this.name, this.value)"/>
    <input type="button" id="id3" name="Right Hand Button" value="Right"
onclick ="buttonReport(this.id, this.name, this.value)"/>
</body>
</html>
```

Use your editor to create the file buttons.html and enter the preceding code. You should find that it generates output messages like the one shown in Figure 3.2, but with different message content depending on which button has been clicked.



```
Button id: id2
Button name: Center Button
Button value: Center
```

FIGURE 3.2 Using a function to send messages

Returning Values from Functions

OK, now you know how to pass information to functions so that they can act on that information for you. But how can you get information back from your function? You won't always want your functions to be limited to popping open a dialog!

Luckily, there is a mechanism to collect data from a function call—the *return value*. Let's see how it works:

```
function cube(x) {
    return x * x * x;
}
```

Instead of using an `alert()` dialog within the function, as in the previous example, this time we prefixed our required result with the `return` keyword. To access this value from outside the function, we simply assign to a variable the value *returned* by the function:

```
var answer = cube(3);
```

The variable `answer` will now contain the value 27.

Note

The values returned by functions are not restricted to numerical quantities as in this example. In fact, functions can return values having any of the data types supported by JavaScript. We discuss data types in Hour 5.

Tip

Where a function returns a value, we can use the function call to pass the return value directly to another statement in our code. For example, instead of

```
var answer = cube(3);
alert(answer);
```

we could simply use

```
alert(cube(3));
```

The value of 27 returned from the function call `cube(3)` immediately becomes the argument passed to the `alert()` method.

Scope of Variables

We have already seen how to declare variables with the `var` keyword. There is a golden rule to remember when using functions:

"Variables declared inside a function only exist inside that function."

This limitation is known as the *scope* of the variable. Let's see an example:

Click here to view code image

```
// Define our function addTax()
function addTax(subtotal, taxRate) {
    var total = subtotal * (1 + (taxRate/100));
    return total;
}
// now let's call the function
var invoiceValue = addTax(50, 10);
alert(invoiceValue); // works correctly
alert(total);  // doesn't work
```

If we run this code, we first see an `alert()` dialog with the value of the variable `invoiceValue` (which should be 55, but in fact will probably be something like 55.000000001 because we have not asked JavaScript to round the result).

We will not, however, then see an `alert()` dialog containing the value of the variable `total`. Instead, JavaScript simply produces an error. Whether you see this error reported depends on your browser settings—you learn more about error handling later in the book—but JavaScript will be unable to display an `alert()` dialog with the value of your variable `total`.

This is because we placed the declaration of the variable `total` *inside* the `addTax()` function. Outside the function the variable `total` simply doesn't exist (or, as JavaScript puts it, "is not defined"). We used the `return` keyword to pass back just the *value* stored in the variable `total`, and that value we then stored in another variable, `invoice`.

We refer to variables declared inside a function definition as being *local* variables; that is, *local to that function*. Variables declared outside any function are known as *global* variables. To add a little more confusion, local and global variables can have the same name, but still be different variables!

The range of situations where a variable is defined is known as the *scope* of the variable—we can refer to a variable as having *local scope* or *global scope*.

Try it Yourself: Demonstrating the Scope of Variables

To illustrate the issue of a variable's scope, take a look at the following piece of code:

Click here to view code image

```
var a = 10;
var b = 10;
function showVars() {
    var a = 20; // declare a new local variable 'a'
    b = 20;     // change the value of global variable 'b'
    return "Local variable 'a' = " + a + "\nGlobal variable 'b' = " +
b;
}
var message = showVars();
alert(message + "\nGlobal variable 'a' = " + a);
```

Within the `showVars()` function we manipulate two variables, `a` and `b`. The variable `a` we define inside the function; this is a local variable that only exists inside the function, quite separate from the global variable (also called `a`) that we declare at the very beginning of the script.

The variable `b` is not declared inside the function, but outside; it is a *global* variable.

Listing 3.3 shows the preceding code within an HTML page.

LISTING 3.3 Global and Local Scope

Click here to view code image

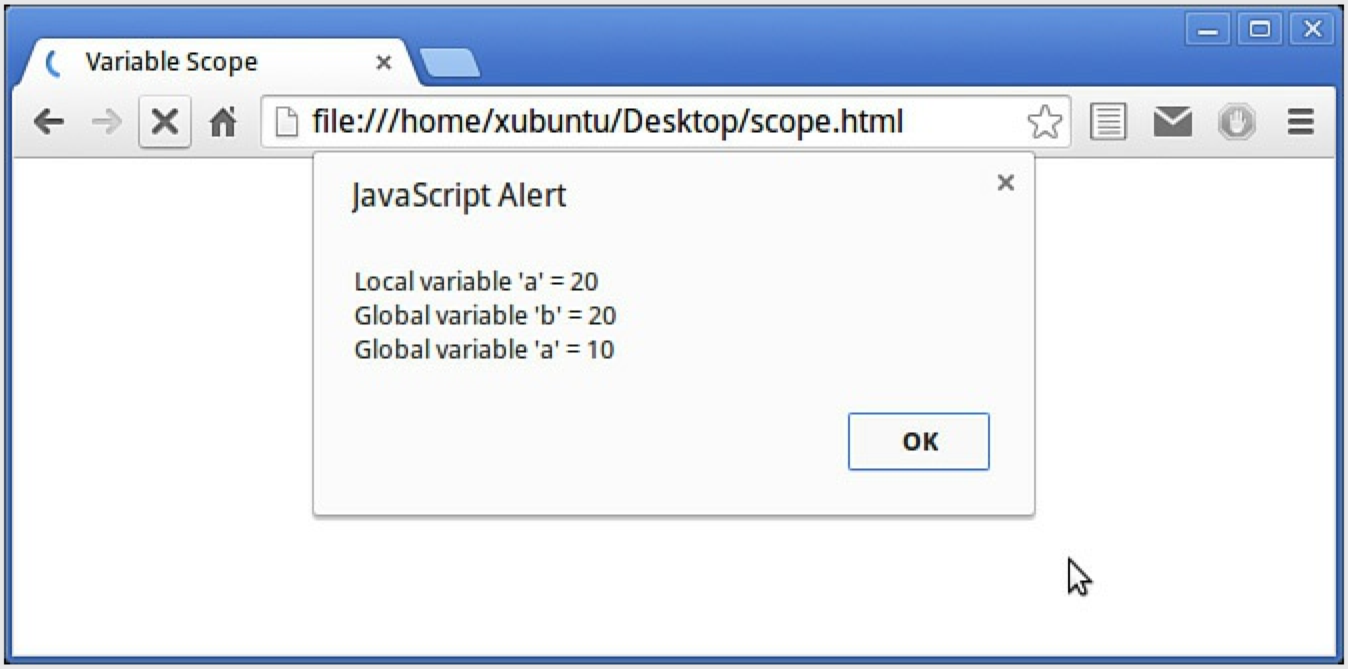
```
<!DOCTYPE html>
<html>
<head>
    <title>Variable Scope</title>
```

```
</head>
<body>
    <script>
        var a = 10;
        var b = 10;
        function showVars() {
            var a = 20; // declare a new local variable 'a'
            b = 20;     // change the value of global variable 'b'
            return "Local variable 'a' = " + a + "\nGlobal variable 'b' =
" + b;
        }
        var message = showVars();
        alert(message + "\nGlobal variable 'a' = " + a);
    </script>
</body>
</html>
```

When the page is loaded, `showVars()` returns a message string containing information about the updated values of the two variables `a` and `b`, as they exist inside the function—`a` with local scope, and `b` with global scope.

A message about the current value of the other, *global* variable `a` is then appended to the message, and the message displayed to the user.

Copy the code into the file scope.html and load it into your browser. Compare your results with Figure 3.3.



```
Local variable 'a' = 20
Global variable 'b' = 20
Global variable 'a' = 10
```

FIGURE 3.3 Local and global scope

Summary

In this hour you learned about what functions are, and how to create them in JavaScript.

You learned how to call functions from within your code, and pass information to those functions in the form of arguments. You also found out how to return information from a function to its calling statement.

Finally, you learned about the local or global scope of a variable, and how the scope of variables affects how functions work with them.

Q&A

Q. Can one function contain a call to another function?

A. Most definitely; in fact, such calls can be nested as deeply as you need them to be.

Q. What characters can I use in function names?

A. Function names must start with a letter or an underscore and can contain letters, digits, and underscores in any combination. They cannot contain spaces, punctuation, or other special characters.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. Functions are called using
   - **a.** The `function` keyword
   - **b.** The `call` command
   - **c.** The function name, with parentheses
2. What happens when a function executes a return statement?
   - **a.** An error message is generated.
   - **b.** A value is returned and function execution continues.
   - **c.** A value is returned and function execution stops.
3. A variable declared inside a function definition is called
   - **a.** A local variable
   - **b.** A global variable
   - **c.** An argument

Answers

1. c. A function is called using the function name.
2. c. After executing a return statement, a function returns a value and then ceases function execution.

3. a. A variable defined within a function has local scope.

Exercises

- Write a function to take a temperature value in Celsius as an argument, and return the equivalent temperature in Fahrenheit, basing it on the code from Hour 2.
- Test your function in an HTML page.

Hour 4. DOM Objects and Built-in Objects

What You'll Learn in This Hour:

- Talking to the user with `alert()`, `prompt()`, and `confirm()`
- Selecting page elements with `getElementById()`
- Accessing HTML content with `innerHTML`
- How to use the browser `history` object
- Reloading or redirecting the page using the `location` object
- Getting browser information via the `navigator` object
- Manipulating dates and times with the `Date` object
- Calculations made easier with the `Math` object

In Hour 1, "Introducing JavaScript," we talked a little about the DOM and introduced the top-level object in the DOM tree, the `window` object. We also looked at one of its child objects, `document`.

In this hour, we introduce some more of the utility objects and methods that you can use in your scripts.

Interacting with the User

Among the methods belonging to the `window` object, there are some designed specifically to help your page communicate with the user by assisting with the input and output of information.

alert()

You've already used the `alert()` method to pop up an information dialog for the user. You'll recall that this modal dialog simply shows your message with a single OK button. The term *modal* means that script execution pauses, and all user interaction with the page is suspended, until the user clears the dialog. The `alert()` method takes a message string as its argument:

```
alert("This is my message");
```

`alert()` does not return a value.

confirm()

The `confirm()` method is similar to `alert()`, in that it pops up a modal dialog with a message for the user. The `confirm()` dialog, though, provides the user with a choice; instead of a single OK button, the user may select between OK and Cancel, as shown in Figure 4.1. Clicking on either button clears the dialog and allows the calling script to continue, but the `confirm()` method returns a different value depending on which button was clicked—Boolean true in the case of OK, or false in the case of Cancel. We begin to look at JavaScript's data types in the next hour, but for the moment you just need to know that a Boolean variable can only take one of two values, *true* or *false*.

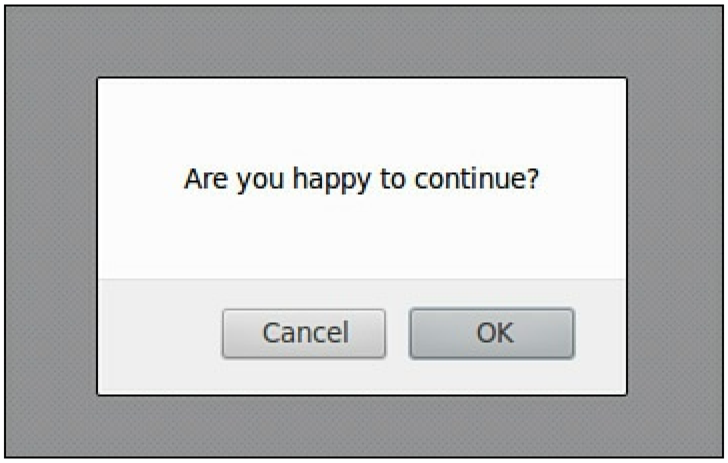

Are you happy to continue?

FIGURE 4.1 The `confirm()` dialog

The `confirm()` method is called in a similar way to `alert()`, passing the required message as an argument:

Click here to view code image

```
var answer = confirm("Are you happy to continue?");
```

Note that here, though, we pass the returned value of `true` or `false` to a variable so we can later test its value and have our script take appropriate action depending on the result.

prompt()

The `prompt()` method is yet another way to open up a modal dialog. In this case, though, the dialog invites the user to enter information.

A `prompt()` dialog is called in just the same manner as `confirm()`:

Click here to view code image

```
var answer = prompt("What is your full name?");
```

The `prompt` method also allows for an optional second argument, giving a default response in case the user clicks OK without typing anything:

Click here to view code image

```
var answer = prompt("What is your full name?", "John Doe");
```

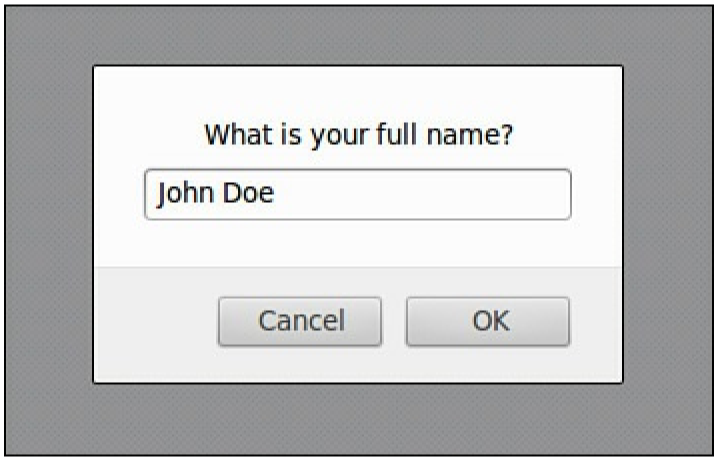
The return value from a `prompt()` dialog depends on what option the user takes:

- If the user types in input and clicks OK or presses Enter, the user input string is returned.
- If the user clicks OK or presses Enter without typing anything into the prompt dialog, the method returns the default response (if any), as optionally specified in the second argument passed to `prompt()`.
- If the user dismisses the dialog (that is, by clicking Cancel or pressing Escape), then the prompt method returns *null*.

Note

The `null` value is used by JavaScript on certain occasions to denote an empty value. When treated as a number it takes the value 0, when used as a string it evaluates to the empty string (“”), and when used as a Boolean value it becomes *false*.

The `prompt()` dialog generated by the previous code snippet is shown in Figure 4.2.


What is your full name?

John Doe

FIGURE 4.2 The `prompt()` dialog

Selecting Elements by Their ID

In Part III, “Objects,” you’ll learn a lot about navigating around the DOM using the various methods of the `document` object. For now, we limit ourselves to looking at one in particular—the `getElementById()` method.

To select an element of your HTML page having a specific ID, all you need to do is call the `document` object’s `getElementById()` method, specifying as an argument the ID of the required element. The method returns the DOM object corresponding to the page element with the specified ID.

Let’s look at an example. Suppose your web page contains a `<div>` element:

Click here to view code image

```
<div id="div1">
    ... Content of DIV element ...
</div>
```

In your JavaScript code, you can access this `<div>` element using `getElementById()`, passing the required ID to the method as an argument:

Click here to view code image

```
var myDiv = document.getElementById("div1");
```

We now have access to the chosen page element and all of its properties and methods.

Caution

Of course, for this to work the page element must have its ID attribute set. Because ID values of HTML page elements are required to be unique, the method should always return a single page element, provided a matching ID is found.

The **innerHTML** Property

A handy property that exists for many DOM objects, `innerHTML` allows us to get or set the value of the HTML content inside a particular page element. Imagine your HTML contains the following element:

Click here to view code image

```
<div id="div1">
    <p>Here is some original text.</p>
</div>
```

We can access the HTML content of the `<div>` element using a combination of `getElementById()` and `innerHTML`:

Click here to view code image

```
var myDivContents = document.getElementById("div1").innerHTML;
```

The variable `myDivContents` will now contain the string value:

Click here to view code image

```
"<p>Here is some original text.</p>"
```

We can also use `innerHTML` to *set* the contents of a chosen element:

Click here to view code image

```
document.getElementById("div1").innerHTML =
    "<p>Here is some new text instead!</p>";
```

Executing this code snippet erases the previous HTML content of the `<div>` element and replaces it with the new string.

Accessing Browser History

The browser's history is represented in JavaScript by the `window.history` object, which is essentially a list of the URLs previously visited. Its methods enable you to use the list, but not to manipulate the URLs explicitly.

The only property owned by the `history` object is its length. You can use this property to find how many pages the user has visited:

Click here to view code image

```
alert("You've visited " + history.length + " web pages in this browser
session");
```

The `history` object has three methods.

`forward()` and `back()` are equivalent to pressing the Forward and Back buttons on the browser; they take the user to the next or previous page in the history list.

```
history.forward();
```

There is also the method `go`, which takes a single parameter. This can be an integer, positive or negative, and it takes the user to a relative place in the history list:

Click here to view code image

```
   history.go(-3);  // go back 3 pages
   history.go(2);  // go forward 2 pages
```

The method can alternatively accept a string, which it uses to find the first matching URL in the history list:

Click here to view code image

```
   history.go("example.com");  // go to the nearest URL in the history
                     // list that contains 'example.com'
```

Using the location Object

The `location` object contains information about the URL of the currently loaded page.

We can think of the page URL as a series of parts:

[protocol]//[hostname]:[port]/[pathname][search][hash]

Here's an example URL: http://www.example.com:8080/tools/display.php?section=435#list

The list of properties of the `location` object includes data concerning the various parts of the URL. The properties are listed in Table 4.1.

Property	Description
`location.href`	'http://www.example.com:8080/tools/display.php?section=435#list'
`location.protocol`	'http:'
`location.host`	'www.example.com:8080'
`location.hostname`	'www.example.com'
`location.port`	'8080'
`location.pathname`	'/tools/display.php'
`location.search`	'?section=435'
`location.hash`	'#list'

TABLE 4.1 Properties of the location Object

Navigating Using the location Object

There are two ways to take the user to a new page using the `location` object.

First, we can directly set the `href` property of the object:

Click here to view code image

```
location.href = 'www.newpage.com';
```

Using this technique to transport the user to a new page maintains the original page in the browser's history list, so the user can return simply by using the browser Back button. If you would rather the sending page were removed from the history list and replaced with the new URL, you can instead use the location object's `replace()` method:

Click here to view code image

```
location.replace('www.newpage.com');
```

This replaces the old URL with the new one both in the browser and in the history list.

Reloading the Page

To reload the current page into the browser—the equivalent to having the user click the "reload page" button—we can use the `reload()` method:

```
location.reload();
```

Tip

Using `reload()` without any arguments retrieves the current page from the browser's cache, if it's available there. To avoid this and get the page directly from the server, you can call `reload` with the argument `true`:

```
document.reload(true);
```

Browser Information—The navigator Object

While the `location` object stores information about the current URL loaded in the browser, the `navigator` object's properties contain data about the browser application itself.

Try it Yourself: Displaying Information Using the navigator Object

We're going to write a script to allow you to find out what the `navigator` object knows about your own browsing setup. Use your editor to create the file navigator.html containing the code from Listing 4.1. Save the file and open it in your browser.

LISTING 4.1 Using the navigator Object

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>window.navigator</title>
    <style>
        td {border: 1px solid gray; padding: 3px 5px;}
    </style>
</head>
<body>
    <script>
        document.write("<table>");
        document.write("<tr><td>appName</td><td>"+navigator.appName + "
</td></tr>");
        document.write("<tr><td>appCodeName</td>
<td>"+navigator.appCodeName + "</td></tr>");
        document.write("<tr><td>appVersion</td><td>"+navigator.appVersion
+ "</td></tr>");
        document.write("<tr><td>language</td><td>"+navigator.language + "
</td></tr>");
        document.write("<tr><td>cookieEnabled</td>
<td>"+navigator.cookieEnabled + "</td></tr>");
        document.write("<tr><td>cpuClass</td><td>"+navigator.cpuClass + "
</td></tr>");
        document.write("<tr><td>onLine</td><td>"+navigator.onLine + "</td>
</tr>");
        document.write("<tr><td>platform</td><td>"+navigator.platform + "
</td></tr>");
        document.write("<tr><td>No of Plugins</td>
<td>"+navigator.plugins.length + "</td></tr>");
        document.write("</table>");
```

FIGURE 17.3 An Ajax form using jQuery

Summary

In this hour you took a good look at the basics of Ajax programming and learned how you can use the jQuery library to make the whole process much more slick and straightforward.

Q&A

Q. How did Ajax get its name?

A. Ajax is an acronym for Asynchronous JavaScript And XML. In practice, though, Ajax is by no means limited to returning just XML data.

Q. Do other libraries besides jQuery implement Ajax?

A. Certainly. There are many libraries and frameworks that help you implement Ajax, some popular ones being Dojo, MooTools, and Prototype.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. Which of these will grab element with `id=source` from server file examples.html and insert it into a page element with `id=target`?

   a. `$("#target").load("examples.html #source");`

   b. `$("#source").load("examples.html #target");`

   c. `$(#source).load("examples.html #info");`

2. A function used to process the data returned from an Ajax call is called:

   a. An anonymous function

   b. A callback function

   c. An Ajax request

3. The jQuery `serialize()` method:

   a. Encodes a set of form elements as a string for submission.

   b. Encodes a set of form elements as a JSON object for submission.

   c. Encodes a set of form elements as a JavaScript array for submission.

Answers

1. a. `$("#target").load("examples.html #source");`
2. b. A callback function
3. a. The jQuery `serialize()` method encodes a set of form elements as a string for submission.

Exercises

▶ Upload some plain text in a .txt file to your server. Create an HTML page that uses jQuery's `load()` method to return the text and display it in a `<div>` element of your page.

▶ Amend the code of this hour's "Try It Yourself" exercise to disallow form submission if either data entry field is blank *or* contains data less than four characters long.

Part VI: Advanced Topics

Hour 18. Reading and Writing Cookies

What You'll Learn in This Hour:

- What cookies are
- All about cookie attributes
- How to set and retrieve cookies
- About cookie expiration dates
- How to save multiple data items in a single cookie
- Deleting cookies
- Escaping and unescaping data
- Limitations of cookies

Something that the JavaScript techniques that you have seen so far can't do is transfer information from one page to another. Cookies provide a convenient way to give your web pages the means to store and retrieve small pieces of information on a user's own computer, allowing your website to save details such as a user's preferences or dates of his or her prior visits to your site.

In this hour you learn how to create, save, retrieve, and delete cookies using JavaScript.

What Are Cookies?

The HTTP protocol that you use to load web pages into your browser is a so-called *stateless* protocol. This means that once the server has delivered the requested page to your browser, it considers the transaction complete and retains no memory of it. This makes it difficult to maintain certain sorts of continuity during a browsing session (or between one session and the next) such as keeping track of which information the visitor has already read or downloaded, or of his or her login status to a private area of the site.

Cookies are a way to get around this problem; you could, for example, use cookies to remember a user's last visit, save a list of that user's preferences, or keep track of shopping cart items while he or she continues to shop. Correctly used, cookies can help improve the experience perceived by the user while using your site.

The cookies themselves are small strings of information that can be stored on a user's computer by the web pages he or she visits, to be later read by any other web pages from within the correct domain and path. Cookies are set to expire after a specified length of time.

Caution

Be aware that many users do not allow websites to leave cookies on their computers, so be sure not to make your websites depend on them.

The usual reason is that some websites use cookies as part of advertising systems, using them to track users' online activities with a view to selecting appropriate advertisements. It may be advisable to show an explanation of why you are going to use the cookie and what you'll use it for.

Limitations of Cookies

Your browser may have a limit to how many cookies it can store—normally a few hundred cookies or more. Usually, 20 cookies per domain name are permitted. A total of 4KB of cookie information can be stored for an individual domain.

In addition to the potential problems created by these size limitations, cookies can also vanish from a hard disk for various reasons, such as the cookie's expiry date being reached or the user clearing cookie information or switching browsers. Cookies should therefore never be used to store critical data, and your code should always be written to cope with situations where an expected cookie cannot be retrieved.

The **document.cookie** Property

Cookies in JavaScript are stored and retrieved by using the `cookie` property of the `document` object.

Each cookie is essentially a text string consisting of a name and a value pair, like this:

```
username=sam
```

When a web page is loaded into your browser, the browser marshals all of the cookies available to that page into a single string-like property, which is available as `document.cookie`. Within `document.cookie`, the individual cookies are separated by semicolons:

Click here to view code image

```
username=sam;location=USA;status=fullmember;
```

Tip

I refer to `document.cookie` as a *string-like* property, because it isn't really a string—it just behaves like one when you're trying to extract cookie information, as you see during this hour.

Escaping and Unescaping Data

Cookie values may not include certain characters. Those disallowed include semicolons, commas, and whitespace characters such as space and tab. Before storing data to a cookie, you need to encode the data in such a way that it will be stored correctly.

You can use the JavaScript `escape()` function to encode a value before storing it, and the corresponding `unescape()` function to later recover the original cookie value.

The `escape()` function converts any non-ASCII character in the string to its equivalent two- or four-digit hexadecimal format—so a blank space is converted into `%20`, and the ampersand character (&) to `%26`.

For example, the following code snippet writes out the original string saved in variable `str` followed by its value after applying the `escape()` function:

Click here to view code image

```
var str = 'Here is a (short) piece of text.';
document.write(str + '<br />' + escape(str));
```

The output to the screen would be

Click here to view code image

```
Here is a (short) piece of text.
Here%20is%20a%20%28short%29%20piece%20of%20text.
```

Notice that the spaces have been replaced by `%20`, the opening parenthesis by `%28`, and the closing parenthesis by `%29`.

All special characters, with the exception of *, @, -, _, +, ., and /, are encoded.

Cookie Ingredients

The cookie information in `document.cookie` may look like a simple string of name and value pairs, each in the form of

```
name=value;
```

but really each cookie has certain other pieces of information associated with it, as outlined in the following sections.

Note

The definitive specification for cookies was published in 2011 as RFC6265. You can read it at http://tools.ietf.org/html/rfc6265.

---

cookieName and cookieValue

These are the name and value visible in each `name=value` pair in the cookie string.

domain

The `domain` attribute tells the browser to which domain the cookie belongs. This attribute is optional, and when not specified its value defaults to the domain of the page setting the cookie.

The purpose of the `domain` attribute is to control cookie operation across subdomains. If the domain is set to www.example.com, then pages on a subdomain such as code.example.com cannot read the cookie. If, however, `domain` is set to example.com, then pages in code.example.com will be able to access it.

You cannot set the `domain` attribute to any domain outside the one containing your page.

path

The `path` attribute lets you specify a directory where the cookie is available. If you want the cookie to be only set for pages in directory `documents`, set the path to `/documents`. The `path` attribute is optional, the usual default path being /, in which case the cookie is valid for the whole domain.

secure

The optional and rarely used `secure` flag indicates that the browser should use SSL security when sending the cookie to the server.

expires

Each cookie has an `expires` date after which the cookie is automatically deleted. The `expires` date should be in UTC time (Greenwich Mean Time, or GMT). If no value is set for `expires`, the cookie will only last as long as the current browser session and will be automatically deleted when the browser is closed.

Writing a Cookie

To write a new cookie, you simply assign a value to `document.cookie` containing the attributes required:

Click here to view code image

```
document.cookie = "username=sam;expires=15/06/2013 00:00:00";
```

To avoid having to set the date format manually, we could do the same thing using JavaScript’s `Date` object:

Click here to view code image

```
    var cookieDate = new Date ( 2013, 05, 15 );
    document.cookie = "username=sam;expires=" + cookieDate.toUTCString();
```

This produces a result identical to the previous example.

Tip

Note the use of

```
cookieDate.toUTCString();
```

instead of

```
cookieDate.toString();
```

because cookie dates always need to be set in UTC time.

In practice, you should use `escape()` to ensure that no disallowed characters find their way into the cookie values:

Click here to view code image

```
var cookieDate = new Date ( 2013, 05, 15 );
var user = "Sam Jones";
document.cookie = "username=" + escape(user) + ";expires=" +
cookieDate.toUTCString();
```

A Function to Write a Cookie

It's fairly straightforward to write a function to write your cookie for you, leaving all the escaping and the wrangling of optional attributes to the function. The code for such a function appears in Listing 18.1.

LISTING 18.1 Function to Write a Cookie

Click here to view code image

```
function createCookie(name, value, days, path, domain, secure) {
    if (days) {
        var date = new Date();
        date.setTime(date.getTime() + (days*24*60*60*1000));
        var expires = date.toGMTString();
    }
    else var expires = "";
    cookieString = name + "=" + escape (value);
    if (expires) cookieString += "; expires=" + expires;
    if (path) cookieString += "; path=" + escape (path);
    if (domain) cookieString += "; domain=" + escape (domain);
    if (secure) cookieString += "; secure";
    document.cookie = cookieString;
}
```

The operation of the function is straightforward. The `name` and `value` arguments are assembled into a `name=value` string, after escaping the `value` part to avoid errors with any disallowed characters.

Instead of specifying a date string to the function, we are asked to pass the number of days required before expiry. The function then handles the conversion into a suitable date string.

The remaining attributes are all optional and are appended to the string only if they exist as arguments.

Caution

Your browser security may prevent you from trying out the examples in this hour if you try simply loading the files from your local machine into your browser. To see the examples working, you may need to upload the files to a web server on the Internet or elsewhere on your local network.

Try it Yourself: Writing Cookies

Let's use this function to set the values of some cookies. The code for our simple page is shown in Listing 18.2. Create a new file named testcookie.html and enter the code as listed. Feel free to use different values for the name and value pairs that you store in your cookies.

LISTING 18.2 Writing Cookies

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
<title>Using Cookies</title>
<script>
    function createCookie(name, value, days, path, domain, secure) {
    if (days) {
            var date = new Date();
            date.setTime(date.getTime() + (days*24*60*60*1000));
            var expires = date.toGMTString();
        }
        else var expires = "";
        cookieString = name + "=" + escape (value);
        if (expires) cookieString += "; expires=" + expires;
        if (path) cookieString += "; path=" + escape (path);
        if (domain) cookieString += "; domain=" + escape (domain);
        if (secure) cookieString += "; secure";
        document.cookie = cookieString;
    }
    createCookie("username","Sam Jones", 5);
    createCookie("location","USA", 5);
    createCookie("status","fullmember", 5);
</script>
</head>
<body>
Check the cookies for this domain using your browser tools.
</body>
</html>
```

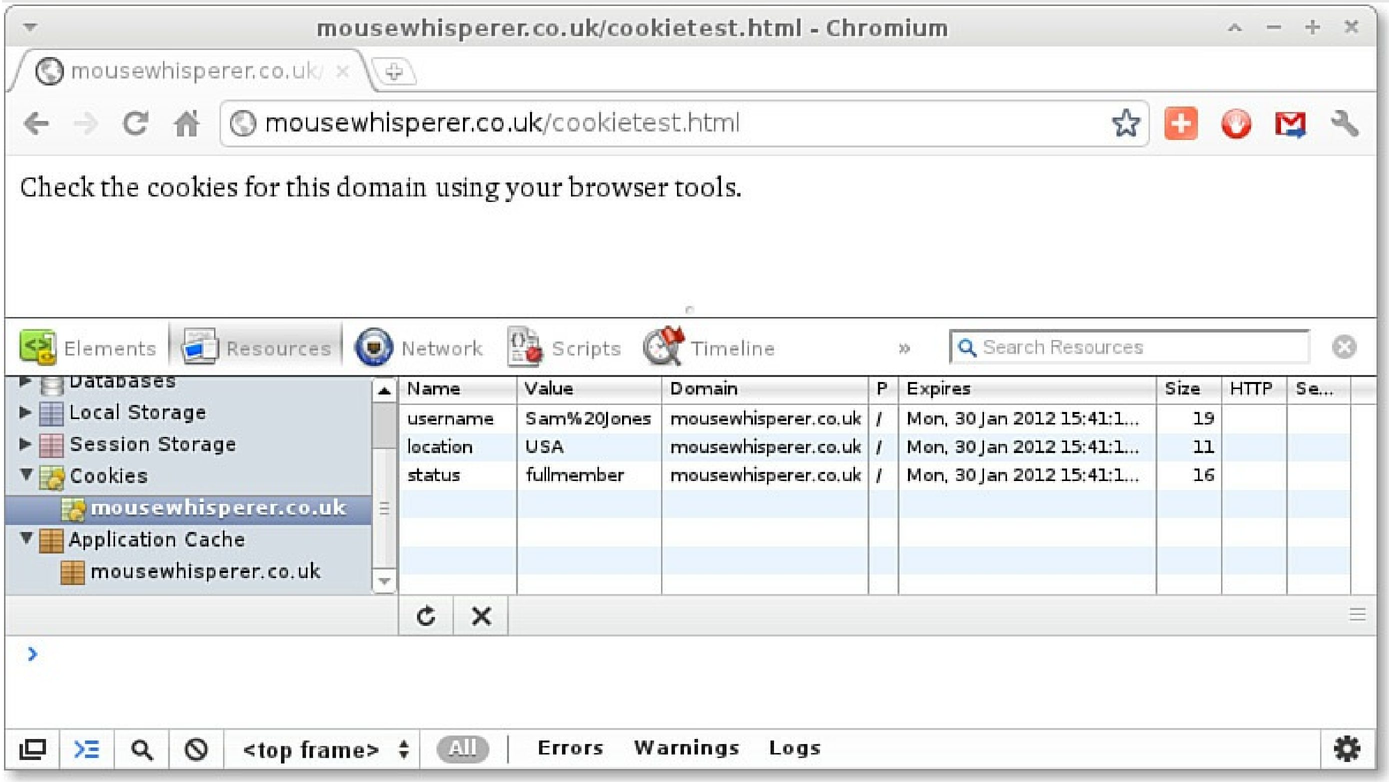
Upload this HTML file to an Internet host or a web server on your local area network, if you have one. The loaded page displays nothing but a single line of text:

Click here to view code image

```
Check the cookies for this domain using your browser tools.
```

In the Chromium browser, I can open Developer Tools using Shift+Ctrl+I—if you are using a different browser, check the documentation for how to view cookie information.

My result is shown in Figure 18.1.



mousewhisperer.co.uk/cookietest.html - Chromium

mousewhisperer.co.uk/cookietest.html

Check the cookies for this domain using your browser tools.

Name	Value	Domain	P	Expires	Size	HTTP	Se...
username	Sam%20Jones	mousewhisperer.co.uk	/	Mon, 30 Jan 2012 15:41:1...	19		
location	USA	mousewhisperer.co.uk	/	Mon, 30 Jan 2012 15:41:1...	11		
status	fullmember	mousewhisperer.co.uk	/	Mon, 30 Jan 2012 15:41:1...	16		

FIGURE 18.1 Displaying our cookies

Tip

Note that each time the function is called, it sets a new value for `document.cookie`, yet this value does not overwrite the previous one;

instead, it appends your new cookie to the cookie values already present. As I said, `document.cookie` sometimes *appears* to act like a string, but it isn't one really.

Reading a Cookie

The function to read the value of a cookie relies heavily on JavaScript's `split()` string method that you learned about in Hour 5, "Numbers and Strings." You may recall that `split()` takes a string and splits it into an array of items, using a specified character to determine where the string should be divided:

Click here to view code image

```
myString = "John#Paul#George#Ringo";
var myArray = myString.split('#');
```

The preceding statement would divide string `myString` into a series of separate parts, cutting the string at each occurrence of the hash (#) character; `myArray[0]` would contain "John," `myArray[1]` would contain "Paul," and so forth.

Since in `document.cookie` the individual cookies are divided by the semicolon character, this character is initially used to break up the string returned by `document.cookie`:

Click here to view code image

```
var crumbs = document.cookie.split(';');
```

You want to search for a cookie of a specific name, so the resulting array `crumbs` is next searched for any items having the appropriate `name=` part.

The `indexOf()` and `substring()` methods are combined to return the value part of the cookie, which is then returned by the function after using `unescape()` to remove any encoding:

Click here to view code image

```
function getCookie(name) {
    var nameEquals = name + "=";
    var crumbs = document.cookie.split(';');
    for (var i = 0; i < crumbs.length; i++) {
        var crumb = crumbs[i];
        if (crumb.indexOf(nameEquals) == 0) {
            return unescape(crumb.substring(nameEquals.length,
crumb.length));
        }
    }
    return null;
}
```

Deleting Cookies

To delete a cookie, all that is required is to set it with an expiry date before the current day. The browser infers that the cookie has already expired and deletes it.

```
function deleteCookie(name) {
    createCookie(name,"",-1);
}
```

Caution

Some versions of some browsers maintain the cookie until you restart your browser even if you have deleted it in the script. If your program depends on the deletion definitely having happened, do another `getCookie` test on the deleted cookie to make sure it has really gone.

Try it Yourself: Using Cookies

Let’s put together all you’ve learned so far about cookies by building some pages to test cookie operation.

First, collect the functions `createCookie()`, `getCookie()`, and `deleteCookie()` into a single JavaScript file and save it as cookie.js, using the code in Listing 18.3.

LISTING 18.3 cookies.js

Click here to view code image

```
function createCookie(name, value, days, path, domain, secure) {
    if (days) {
        var date = new Date();
        date.setTime( date.getTime() + (days*24*60*60*1000));
        var expires = date.toGMTString();
    }
    else var expires = "";
    cookieString = name + "=" + escape (value);
    if (expires) cookieString +=   "; expires=" + expires;
    if (path) cookieString += "; path=" + escape (path);
    if (domain) cookieString += "; domain=" + escape (domain);
    if (secure) cookieString += "; secure";
    document.cookie = cookieString;
}

function getCookie(name) {
    var nameEquals = name + "=";
    var crumbs = document.cookie.split(';');
    for (var i = 0; i < crumbs.length; i++) {
        var crumb = crumbs[i].trim();
        if (crumb.indexOf(nameEquals) == 0) {
            return unescape(crumb.substring(nameEquals.length,
crumb.length));
        }
    }
    return null;
}

function deleteCookie(name) {
    createCookie(name,"",-1);
}
```

This file will be included in the `<head>` of your test pages so that the three functions are available for use by your code.

The code for the first test page, cookietest.html, is listed in Listing 18.4, and that for a second test page, cookietest2.html, in Listing 18.5. Create both of these pages in your text editor.

LISTING 18.4 cookietest.html

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Cookie Testing</title>
    <script src="cookies.js"></script>
    <script>
        window.onload = function() {
            var cookievalue = prompt("Cookie Value:");
            createCookie("myCookieData", cookievalue);
        }
    </script>
</head>
<body>
    <a href="cookietest2.html">Go to Cookie Test Page 2</a>
</body>
</html>
```

LISTING 18.5 cookietest2.html

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Cookie Testing</title>
    <script src="cookies.js"></script>
    <script>
        window.onload = function() {
            document.getElementById("output").innerHTML = "Your cookie
value: " + getCookie("myCookieData");
        }
    </script>
</head>
<body>
    <a href="cookietest.html">Back to Cookie Test Page 1</a><br/>
    <div id="output"></div>
</body>
</html>
```

The only visible page content in cookietest.html is a link to the second page cookietest2.html. However, the `window.onload` event is captured by the code on the page and used to execute a function that launches a `prompt()` dialog as soon as the page has finished loading. The dialog asks you for a value to be saved to your cookie, and then calls `createCookie()` to set a cookie of name `myCookieData` with the value that you just entered.

The page cookietest.html is shown working in Figure 18.2.

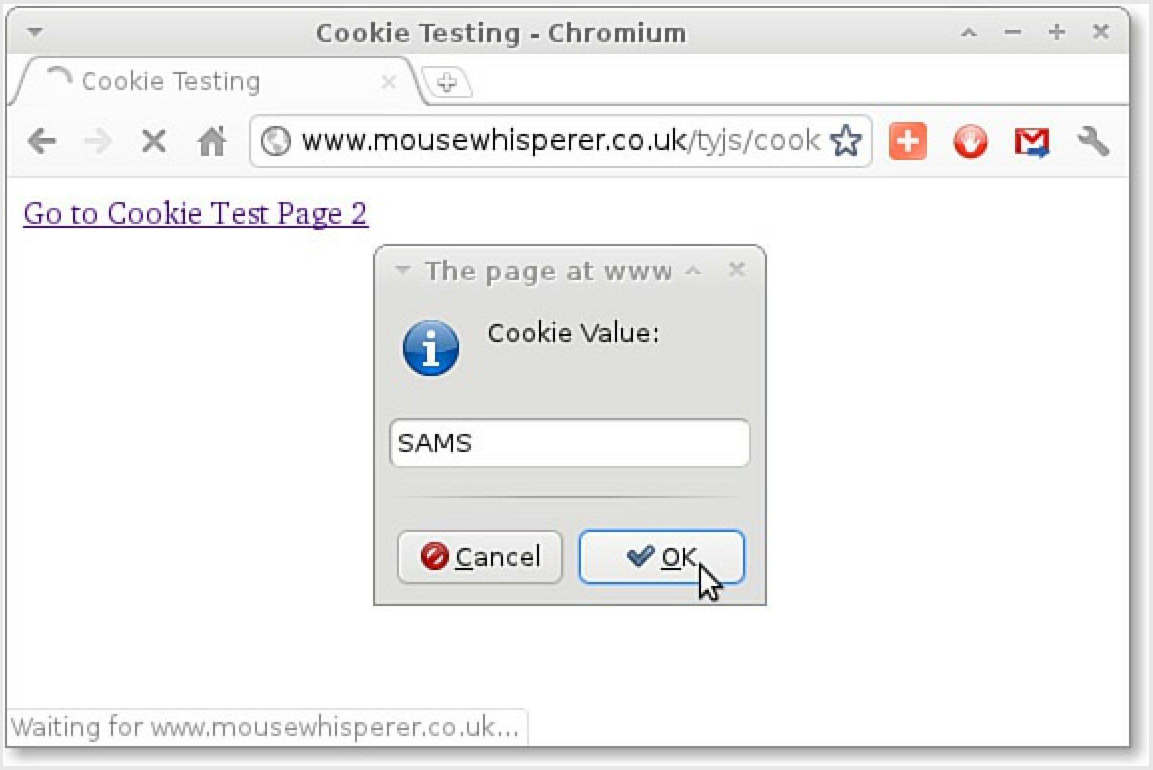
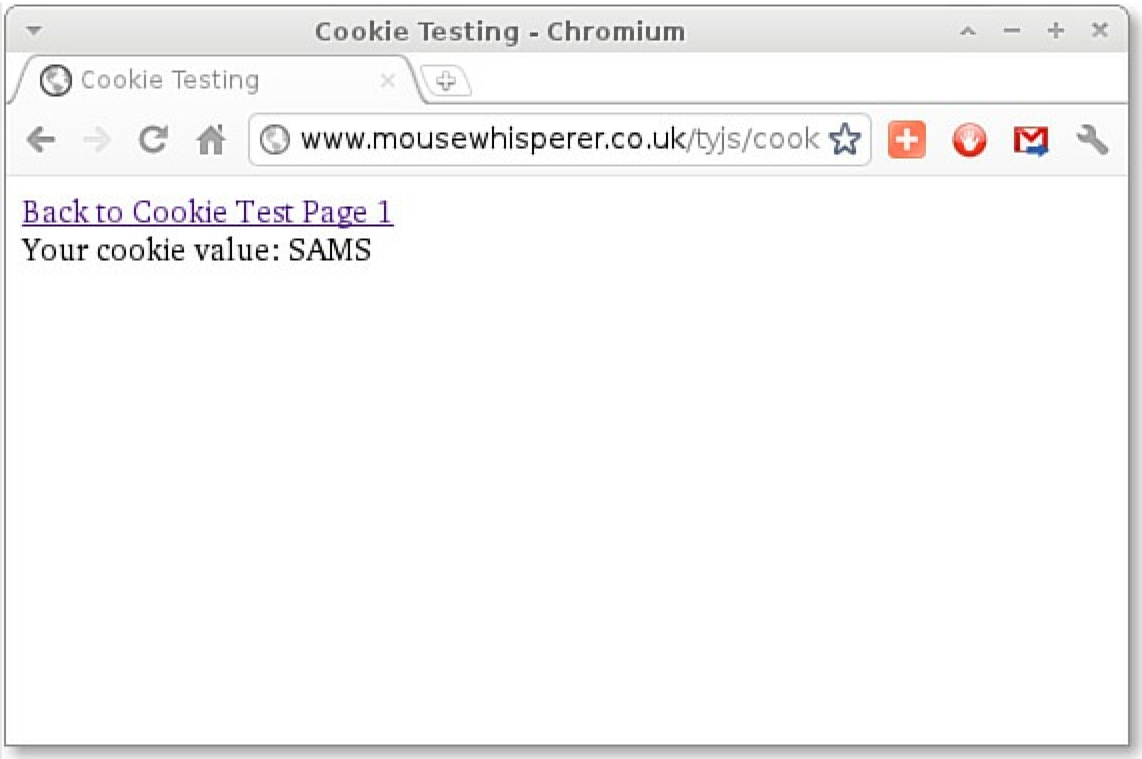

Cookie Testing - Chromium
Cookie Testing
www.mousewhisperer.co.uk/tyjs/cook
Go to Cookie Test Page 2
The page at www
Cookie Value:
SAMS
Cancel
OK
Waiting for www.mousewhisperer.co.uk...

FIGURE 18.2 Enter a value for your cookie.

After setting your cookie, use the link to navigate to cookietest2.html.

When this page loads, the `window.onload` event handler executes a function that retrieves the stored cookie value using `getCookie()` and writes it to the page, as shown in Figure 18.3.



Back to Cookie Test Page 1
Your cookie value: SAMS

FIGURE 18.3 Retrieving the value of your cookie

To try it out for yourself, you need to upload the files cookietest.html, cookietest2.html, and cookies.js to a web server on the Internet (or one on your local network, if you have one) as browser security will probably prevent you from setting cookies when using the `file://` protocol to view a file on your own computer.

Setting Multiple Values in a Single Cookie

Each cookie contains one `name=value` pair, so if you need to store several separate pieces of data such as a user's name, age, and membership number, you need three different cookies.

However, with a little ingenuity you can make your cookie store all three values by concatenating the required values into a single string, which becomes the value stored by your cookie.

This way, instead of having three separate cookies for name, age, and membership number, you could have just one, perhaps named `user`, containing all three pieces of data. To separate the details later, you place in your value string a special character called a *delimiter* to separate the different pieces of data:

Click here to view code image

```
var userdata = "Sandy|26|A23679";
createCookie("user", userdata);
```

Here the | (pipe) character acts as the delimiter. When you later retrieve the cookie value, you can split it into its separate variable values by using the | delimiter:

Click here to view code image

```
var myUser = getCookie("user");
var myUserArray = myUser.split('|');
var name = myUserArray[0];
var age = myUserArray[1];
var memNo = myUserArray[2];
```

Cookies that store multiple values use up fewer of the 20 cookies per domain allowed by some browsers, but remember that your use of cookies is still subject to the 4KB overall limit for cookie information.

Note

This is a further example of serialization, which you learned about in Hour 10, "Meet JSON."

Summary

In this hour you learned about cookies, and how to set, retrieve, and delete them using JavaScript. You also learned how to concatenate multiple values into a single cookie.

Q&A

Q. When concatenating multiple values into a single cookie, can you use any character as a delimiter?

A. You can't use any character that might appear in your escaped data (except as the delimiter character), nor can you use equals (=) or the semicolon (;) as these are used to assemble and concatenate the `name=value` pairs in `document.cookie`. Additionally, cookies may not include whitespace or commas, so naturally they cannot be used as delimiters either.

Q. Are cookies safe?

A. Questions are often raised over the security of cookies, but such fears are largely unfounded. Cookies *can* help website owners and advertisers track your browsing habits, and they can (and do) use such information to select advertisements and promotions to show on web pages that you visit. Website owners and advertisers can't, however, find out personal information about you or access other items on your hard disk simply through the use of cookies.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. Cookies are small pieces of text information stored
   a. On a user's hard disk
   b. On the server
   c. At the user's Internet service provider
2. Encoding a string to store it safely in a cookie can be carried out by using
   a. `escape()`
   b. `unescape()`
   c. `split()`
3. A character used to separate multiple values in a single cookie is known as
   a. An escape sequence
   b. A delimiter
   c. A semicolon

Answers

1. a. Cookies are stored on a user's hard disk.
2. a. You can use `escape()` to safely encode string values for storage in a cookie.
3. b. Multiple values are separated by a character called a delimiter.

Exercises

- Find out how to view cookie information in your favorite browser. Use the browser tools to examine the cookie set by the code of Listing 18.4.
- Rewrite the code for cookietest.html and cookietest2.html to write multiple values to the same cookie and separate them on retrieval, displaying the values on separate lines. Use the hash character (#) as your delimiter.
- Add a button to cookietest2.html to delete the cookie set in cookietest.html and check that it works as requested. (Hint: Use the button to call `deleteCookie()`.)

Hour 19. Coming Soon to JavaScript

What You'll Learn in This Hour:

- About some of the most important new additions coming soon to JavaScript
- How to find out which features are supported by which browsers
- How to use some of the new language features right away

ECMAScript 6 (codenamed *Harmony*) is the forthcoming version of the ECMAScript standard that underpins the JavaScript language. This new standard should be ratified sometime in 2015.

ECMAScript 6 is a significant update to the specification, and the first major update to the language since ECMAScript 5 became standardized in 2009. The major browser manufacturers are already working on implementing the new features in their JavaScript engines.

In this hour we'll take a look at a few of the most important new features, some of which you can already use.

Tip

At the time of writing, Google's Chrome browser has support for ECMAScript 6 turned off by default. You can turn it on by visiting the chrome://flags/ page and finding the Enable Experimental JavaScript entry.

Note

You can check out the current compatibility status for various browsers and ECMAScript 6 features at http://kangax.github.io/compat-table/es6/.

Classes

In Hour 8, "Object-Oriented Programming," you read about OOP and saw examples of how to create and manipulate objects, including this one:

Click here to view code image

```
function Car(Color, Year, Make, Miles) {
   this.color = Color;
   this.year = Year;
   this.make = Make;
   this.odometerReading = Miles;
   this.setOdometer = function(newMiles) {
```

```
            this.odometerReading = newMiles;
        }
    }
```

If you've come to JavaScript from another programming language you may already be familiar with *classes*. A class is a representation of an object.

Click here to view code image

```
    class Car {
        constructor(Color, Year, Make, Miles) {
            this.color = Color;
            this.year = Year;
            this.make = Make;
            this.odometerReading = Miles;
        }

        setOdometer(newMiles) {
            this.odometerReading = newMiles;
        }
    }
```

This syntax also allows you to *extend* classes, creating a new class that inherits the properties of the parent. Here's an example:

Click here to view code image

```
    class Sportscar extends Car {
        constructor(Color, Year, Make, Miles) {
            super(Color, Year, Make, Miles);
            this.doors = 2;
        }
    }
```

Here I've used the `super` keyword in my constructor, allowing me to call the constructor of a parent class and inherit all of its properties. In truth, this is just syntactic sugar; everything using classes can be rewritten in functions and prototypes, just like you learned in Hour 8. However, it's much more compatible with other popular languages, and somewhat easier to read.

Arrow Functions

The arrow function (=>) is a shorthand syntax for an anonymous function.

Click here to view code image

```
param => statements or expression
```

Let's explicate this a bit more:

- *param*—The name of an argument or arguments. If the function has zero arguments, this needs to be indicated with (). For only one argument the parentheses are not required.
- *statements or expression*—Multiple statements need to be enclosed in curly

braces. A single expression, though, doesn't need braces. The expression is also the return value of that function.

[Click here to view code image]

```
var overTen = x => x > 10 ? 10 : x;
overTen(8); // returns 8
overTen(12); // returns 10
```

Note that the `function` keyword isn't required, and that the parentheses can be omitted since there is a single argument. The following example has two arguments:

[Click here to view code image]

```
var higher = (x, y) => {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
higher(7, 9); // returns 9
higher(12, 3); // returns 12
```

As well as being a little simpler to write, arrow functions also have the feature that they inherit the value of `this` from the container. This is really handy when using objects. Previously we needed to assign `this` to a variable to pass it into a function:

[Click here to view code image]

```
function myObject() {
  this.height = 13;
  var self = this;

  setTimeout(function fiveSecondsLater() {
    console.log(self.height);
  }, 5000)
}
var o = new myObject();
```

In the preceding example, we couldn't simply use

```
console.log(this.height);
```

because `this` would refer to its immediate container, here the function `fiveSecondsLater()`. However, by using arrow functions the use of a variable like `self` can be avoided:

[Click here to view code image]

```
function myObject() {
  this.height = 13;

  setTimeout(() => {
    console.log(this.height); // 'this' here refers to myObject
  }, 5000)
}
var o = new myObject();
```

Modules

As JavaScript applications grow in complexity, a means needs to be found to make objects declared in one file available in others. By this means, larger projects can be written in a modular fashion.

By default, anything you declare in one file is not available outside of that file. In ECMAScript 6, though, you can use the `export` keyword to make it available.

Here's an example of how to export a class:

Click here to view code image

```
// this code appears in file1.js
export default function Car(Color, Year, Make, Miles) {
 this.color = Color;
 this.year = Year;
 this.make = Make;
 this.odometerReading = Miles;
 this.setOdometer = function(newMiles) {
    this.odometerReading = newMiles;
 }
 // this object can be imported by other files
```

And in the receiving file:

Click here to view code image

```
//  this code appears in file2.js
import Car from 'file1';
var ferrari = new Car('red', 1986, 'Dino', 75500);
```

Using let and const

Before ECMAScript 6, JavaScript had only two types of scope—namely, *function scope* and *global scope*. (The scope of a variable, as you learned in Hour 3, "Using Functions," depends on whereabouts in the code the variable was declared using the `var` keyword.)

To the frustration of many developers coming to JavaScript from other languages, JavaScript lacked a so-called *block scope*, defining that a variable is only accessible within the block in which it's defined. (A block is everything inside a pair of curly braces.)

The new keyword `let` allows you to declare a variable while limiting its scope to the block, statement, or expression on which it is declared.

The `var` keyword, in contrast, defines a variable either globally or locally to an entire function, taking no account of block scope:

Click here to view code image

```
function myFunc() {
  {
    let x;
    if(y == 0)
      {
         // this is ok, x has block scope
         let x = "inner";
      }
    // this is an error, x already declared in block
    let x = "outer";
  }
}
```

The `const` declaration creates a constant—that is, a read-only named variable. The value of a constant cannot change through reassignment, nor can a constant be re-declared later.

Click here to view code image

```
function myFunc() {
  {
    const x = "foo";

    // this is an error, x is constant, can't be re-defined
    x = "bar";
  }
}
```

Try it Yourself: Checking Out **const**

Let's have a look at how `const` operates. At the time of writing, it works in most browsers, but I'm going to use Google Chrome.

Instead of writing code in a text file, open the JavaScript Console for your browser. In the case of Chrome, I can do that with Ctrl+Shift+J as shown in Figure 19.1.

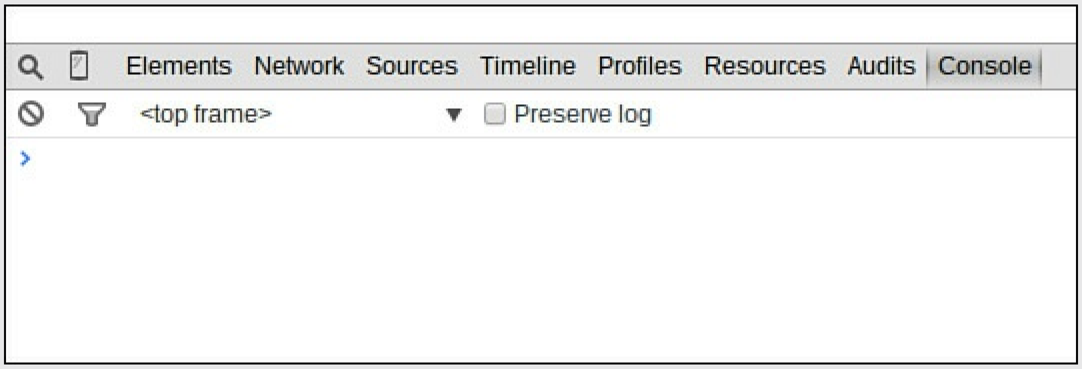
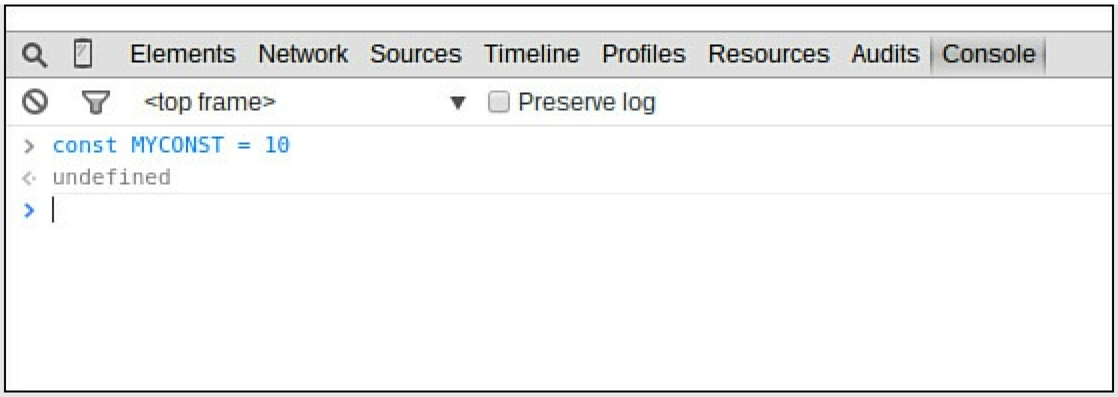


FIGURE 19.1 Chrome's JavaScript Console

First, define a constant using the `const` keyword. You can call it anything you like and choose any value. Mine is called `MYCONST` and I've given it a value of 10 (see Figure 19.2).

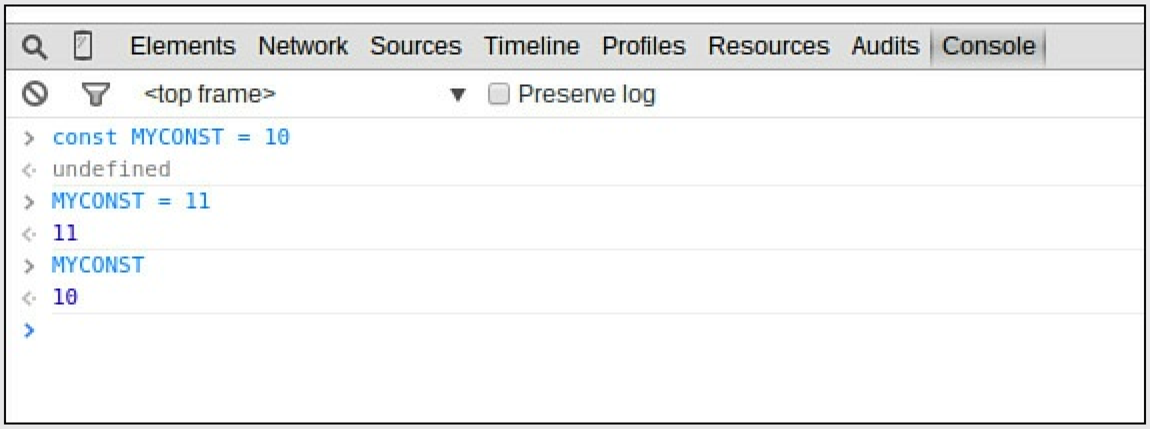


```
> const MYCONST = 10
<· undefined
>
```

FIGURE 19.2 Setting a constant

The console issues undefined because the declaration of a `const` does not return a value.

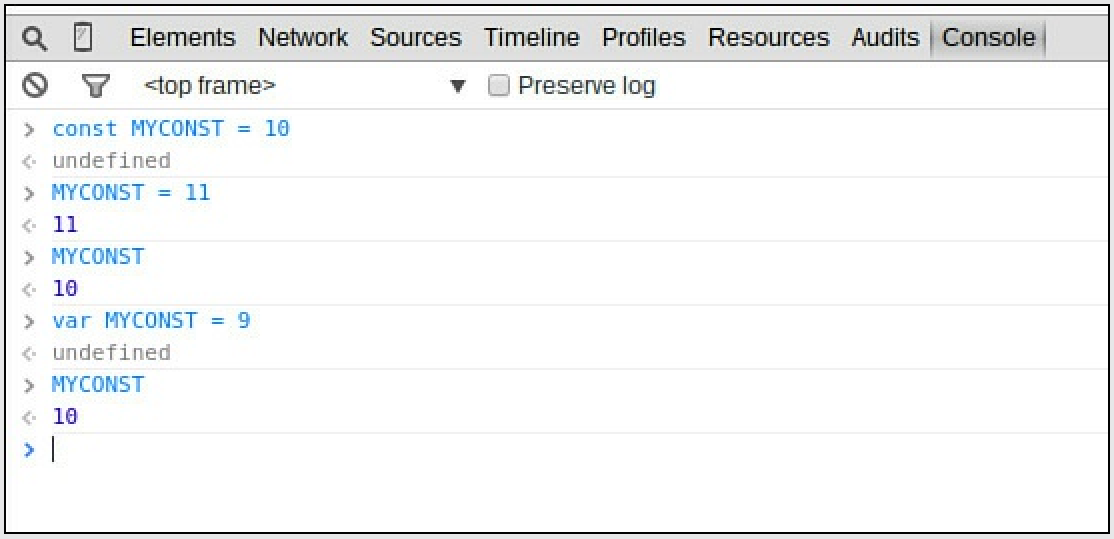
In Figure 19.3 I try to redefine the value of MYCONST.



```
> const MYCONST = 10
<· undefined
> MYCONST = 11
<· 11
> MYCONST
<· 10
>
```

FIGURE 19.3 The constant can’t be reassigned

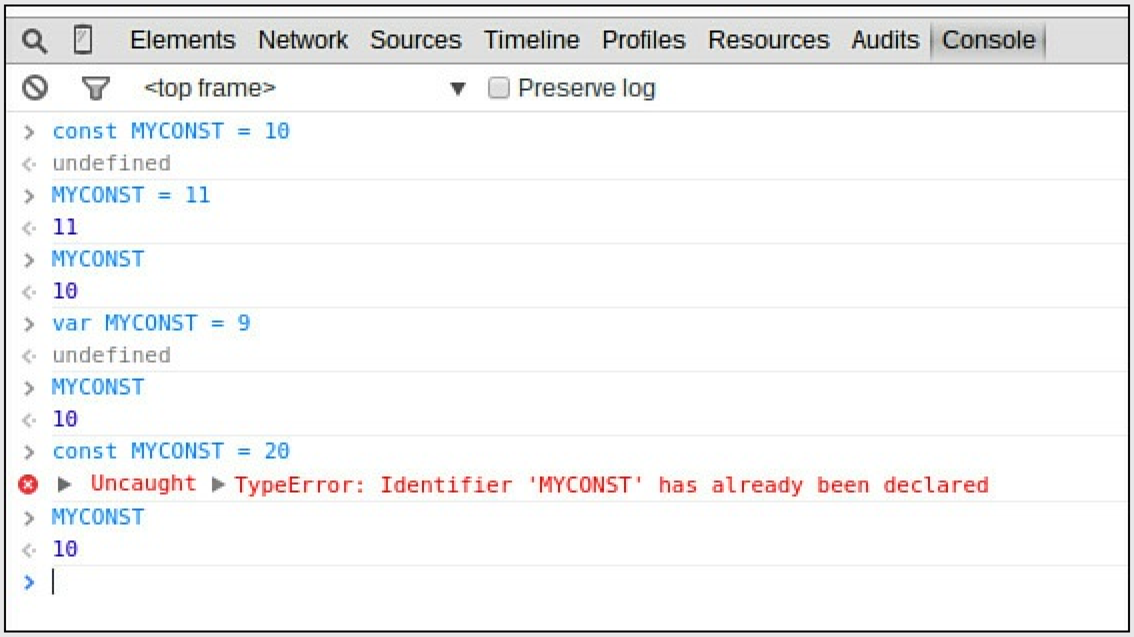
As you can see, the constant `MYCONST` couldn’t be reassigned a new value. Let’s try to re-declare it instead (see Figure 19.4).



```
> const MYCONST = 10
<· undefined
> MYCONST = 11
<· 11
> MYCONST
<· 10
> var MYCONST = 9
<· undefined
> MYCONST
<· 10
>
```

FIGURE 19.4 Redeclaration of a constant doesn’t work

Nope, we can’t do that either. Finally, let’s try to reinitialize it (see Figure 19.5).



```
> const MYCONST = 10
<· undefined
> MYCONST = 11
<· 11
> MYCONST
<· 10
> var MYCONST = 9
<· undefined
> MYCONST
<· 10
> const MYCONST = 20
✖ ▶ Uncaught ▶ TypeError: Identifier 'MYCONST' has already been declared
> MYCONST
<· 10
> |
```

FIGURE 19.5 Trying to reinitialize throws an error

JavaScript throws an error.

Values declared using the `const` keyword, as we can see, cannot be reinitialized, re-declared, or reassigned.

Template Strings

Template strings provide help in constructing strings and are similar to string interpolation features in some other programming languages such as Perl and Python (among others).

Click here to view code image

```
var name = "John";
var course = "Mathematics III";
var myString = 'Hello ${name}, welcome to ${course}.';
```

You can substitute more complex expressions too:

Click here to view code image

```
var total = 20;
var tax = 4;
msg = 'Total is ${total} (or ${total + tax}, including tax)';
alert(msg);        // "Total is 20 (or 24, including tax)"
```

Access Arrays with **for-of**

JavaScript has various methods for handling arrays, as you read about in Hour 6, “Arrays.” Apart from `while` and `for` loops, you can also use `for-in`. Unfortunately, this loop visits all of an array’s *named properties*, not just the actual array *values*:

Click here to view code image

```
"use strict";
let arr1 = [ 6, 5, 7, 9 ];
arr1.greeting = "hi";

for (var x in arr1) {
   console.log(x); // logs "0", "1", "2", "3", "greeting"
}
```

To get around this problem, ECMAScript 6 introduces the `for-of` construct, which iterates over just the property values:

Click here to view code image

```
for (var y of arr1) {
   console.log(y); // logs "6", "5", "7", "9"
}
```

Note

Note the use of the directive `"use strict"` in the preceding code snippet. This directive, introduced in ECMAScript 5, indicates that JavaScript should execute in strict mode, a more rigid set of interpreter rules, and is currently necessary to use certain ECMAScript 6 features.

Transpilation

The examples presented so far in this hour are fine for testing ECMAScript 6 features, but at the time of writing they are not ready for use in your production code. Few visitors to your website will be using a browser with strong ECMAScript 6 support. You *can* start preparing for the future, though.

Traceur is a Google project intended to take ECMAScript 6 code and process it into ECMAScript 5 code that is compatible with most browsers using their default settings. It doesn’t support all of the ECMAScript 6 features, but new features are being added all the time.

You can read about the project at https://code.google.com/p/traceur-compiler/wiki/GettingStarted, and also download the code to try for yourself at https://github.com/google/traceur-compiler.

Summary

In this hour, you’ve read about just some of the important new changes coming to the JavaScript language in the ECMAScript 6 specification.

The new language features bring the JavaScript syntax more into line with other popular languages, as well as making code more concise and readable.

Browser vendors have already begun to implement these and other ECMAScript 6 features into their offerings, and more of the specification will doubtless be supported in upcoming browser versions.

Q&A

Q. How can I follow the progress of the ECMAScript 6 specification?

A. Probably the best online resource is the official ECMAScript wiki (http://wiki.ecmascript.org/).

Q. Who or what is Ecma?

A. Ecma is an international, membership-based, non-profit standards organization, originally called the European Computer Manufacturers Association (ECMA). The organization was founded in 1961 to standardize computer systems throughout Europe.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. A value declared as a `const`:

   a. Can later be reassigned but not re-declared

   b. Can later be re-declared but not reassigned

   c. Can later neither be re-declared nor reassigned

2. You can use the `export` keyword to

   a. Save all your program data into another file

   b. Make something you declare in one file available outside of that file

   c. Make something you declared in another file available inside of the current file

3. Which of the following is a correct arrow function to turn a Centigrade temperature into Fahrenheit?

   a. `var fahr = cent =&gt; cent * 1.8 + 32;`

   b. `var fahr =&gt; cent =&nbsp;cent * 1.8 + 32;`

   c. `function fahr = cent =&gt; cent * 1.8 + 32;`

Answers

1. c. A const can later neither be re-declared nor reassigned.

2. b. Make something you declare in one file available outside of that file

3. a. `var fahr = cent =&gt; cent * 1.8 + 32;`

Exercises

- Check out how many ECMAScript6 features are supported in your current browser, by using one of the online resources mentioned earlier. Write some small code examples to check the operation of any supported features described in this hour.
- Check out the documentation on the official Ecma wiki for the other ECMAScript 6 features not discussed in this hour.

Hour 20. Using Frameworks

What You'll Learn in This Hour:

- What frameworks are, and why they're useful
- About the Model-View-Controller (MVC) architecture
- How to get started with Google's AngularJS framework
- Details of some other popular frameworks

If you've already written a number of applications, chances are you've had to solve some of the same coding problems over and over again. One of the techniques you can use to cut down on such re-invention of the wheel is to use a *software framework*.

In this hour you'll learn about a popular style of network called an MVC (Model-View-Controller) framework, and see how to implement such a framework for single-page JavaScript applications by using Google's AngularJS.

Software Frameworks

The purpose of a framework is to improve the efficiency with which you can write software applications, at the same time adding consistency, quality, reliability, and robustness to your application.

Choosing a well-written and appropriate framework can leave you more time to focus on the unique requirements of your application rather than spending lots of valuable time on the application's infrastructure.

Why Use a Framework?

Frameworks help you to *reuse code* that has been previously built and tested, improving your application's reliability and reducing the coding and testing work required in its creation.

A framework can also encourage better programming practices, due to the structure it imposes on your application.

Finally, a framework usually provides you with the means to extend its functionality, making it better suited to your application's needs.

Frameworks Are Not the Same as Libraries

Many people confuse the term software framework with a software library, like the ones discussed in Part V of this book.

However, there is a fundamental difference between frameworks and libraries; when you use a library, the objects and methods available within that library already exist, waiting to be invoked by your custom application. You need to know which objects and methods to employ in your code in order to create your application.

When you use a framework, it's you that designs and codes the objects and methods used by your application. The framework provides a consistent structure in which you can do this.

Model-View-Controller (MVC) Architecture

The concept of the Model-View-Controller (MVC) software architecture is fairly simple: to separate our application into units, each of which conforms to one of the following parts.

Models

Models represent the part of the application dealing with business data and business logic. A model might be a single object, or it could be some structure made up from a variety of objects.

Views

A view is a representation of a model used to present information to the user. It usually acts as a presentation filter, showing only certain aspects of the data contained in a model while suppressing others.

A view interrogates its model to obtain the data necessary for presentation. It might also change the data in the model by sending appropriate commands. Such questions and commands all have to be in semantics defined within the model.

Controllers

A controller forms the link between the user and the application, arranging for views to be displayed on the screen, or reading user input by presenting menus, input fields, buttons, or other page elements. The controller interprets user input before passing it to one or more of the views.

The operation of the various parts of the MVC framework is shown in Figure 20.1.

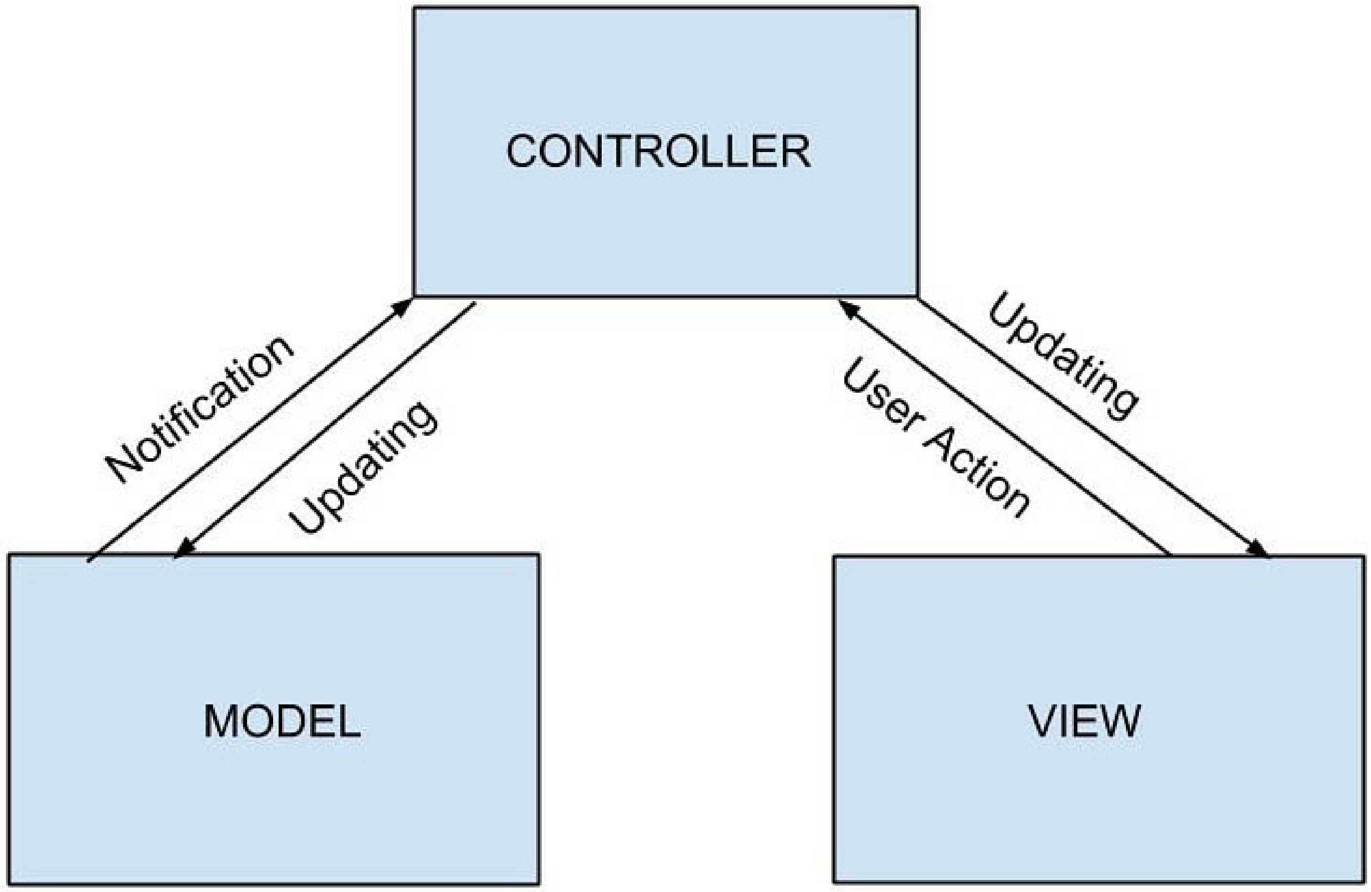


FIGURE 20.1 Model-View-Controller framework interactions

A Real-World Example

Watching TV can be analogous to an MVC framework.

The broadcaster makes available various channels, each containing different data; these channels can be thought of as the *models* of the MVC system.

The *view* is provided by the TV's screen.

You can interact with the TV by using the functions of the remote *control(ler)*.

Using an MVC Framework for Web Apps

The MVC architecture lends itself very well to web applications.

- **Model**—The page content is stored in the models that underpin the application. The technical details may vary—the text and images may be stored in a database, as server files, or in some other way—but the content, and the rules of how it all fits together, are encoded into the model part of the framework.
- **View**—The HTML and CSS add one or more visual display layers to the content —the veneer we apply to give our web application a particular appearance and style. We can change how the content is displayed without altering the original

content, as stored in the model(s), at all.

- **Controller**—The controller element consists of program code linked to the interactive elements on the page, such as form fields, buttons, and links. Such code interprets user input and communicates with models and views.

The AngularJS Framework

AngularJS is an MVC framework developed by Google that lets you build well-structured, easily testable, and maintainable JavaScript web applications. It is designed to help you produce powerful, reliable, single-page web applications.

An Overview of AngularJS

AngularJS is an MVC framework that binds your HTML code (corresponding to the *views* part of the MVC paradigm) to JavaScript objects (the *models* part of MVC).

In one direction, any changes to your models update the page automatically. The opposite is also true—a model, for instance associated with a text field, is updated when the content of the field is changed. In the same manner, any changes in the view—such as a user entering informtion in a field, or clicking on a button—make the required changes to the appropriate model(s).

Behind the scenes, AngularJS handles all the logic, so you don't have to write code to update your page's HTML code, or to listen for and act upon user events.

Including AngularJS in your page

To use AngularJS you have to include it in your page. Perhaps the easiest way to do that is via Google's CDN:

Click here to view code image

```
<script src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/
angular.min.js"></script>
```

Extending HTML with `ng-` directives

AngularJS employs a number of directives that help you associate your page's HTML elements to models in the MVC architecture. These directives each start with `ng-` and can be added to any element.

The key attribute that you have to include in any page is `ng-app`, which defines an AngularJS application. You need to apply this to an element that contains the rest of the page elements bearing `ng-` directives. You can apply it to the page's `<body>` element (making the whole page part of the application), or a `<div>` element enclosing the application:

```
<body ng-app>
```

AngularJS finds this element when the page loads and processes all `ng-` directives it sees on its child elements.

Two further important `ng-` directives are `ng-model` and `ng-bind`.

The `ng-model` directive connects the value of HTML controls such as input fields, select boxes, text areas and so on, to application data stored in models.

The `ng-bind` directive binds that application data, in the MVC models, to elements in the HTML view.

A trivial example is shown in Listing 20.1.

LISTING 20.1 A Simple AngularJS Example

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
<title>AngularJS Example</title>
<style>
 #output {
  font: 28px bold helvetica, arial, sans-serif;
  color: red;
 }
</style>
<script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
</script>
</head>
<body ng-app>
  <p>Name: <input type="text" ng-model="name"></p>
  <span id="output" ng-bind="name"></span>
</body>
</html>
```

AngularJS begins work as soon as the web page has loaded. The `ng-app` directive tells AngularJS that, in this case, it is the `<body>` element of the page that contains an AngularJS application.

The `ng-model` directive then binds the value contained by the input field to the variable `name`.

Similarly, the `ng-bind` directive binds the HTML content of the `<span>` element to the variable `name`. In this way, the `<span>` element becomes a view in our MVC framework.

Now, any changes in the input field will be immediately reflected in the `<span>` element, as shown in Figure 20.2.

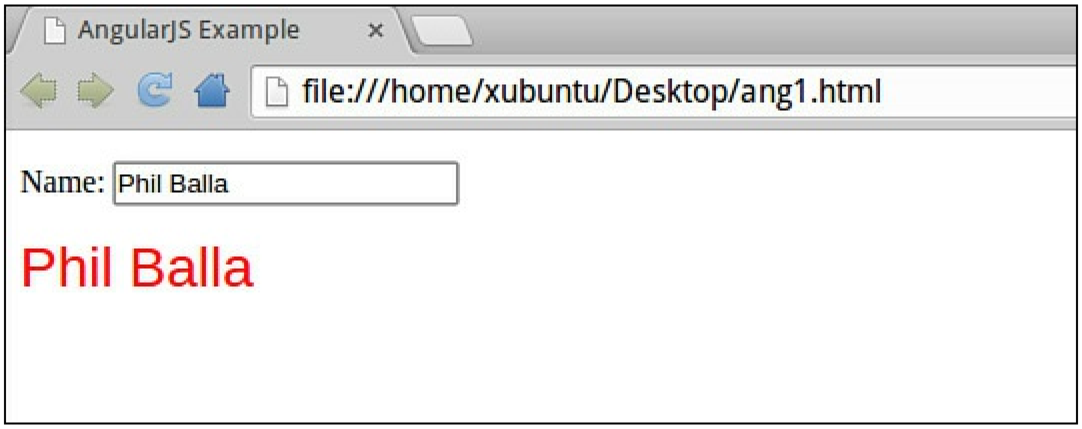


FIGURE 20.2 A simple AngularJS application

That's it. We already have a dynamic application that would ordinarily have taken much more code to build. We didn't have to worry about writing code for data binding and updating, nor specify any models. In fact, we haven't yet written any JavaScript of our own! The application—simple as it is—already works because of AngularJS's design.

Scopes

A scope is an object that links a DOM element (the *view* part of the MVC architecture) to a controller; in MVC terms, this object becomes the *model*.

The controller and the view both have access to the scope model, so it can be used to communicate between them. This scope object will house the data and the methods to be used in the view.

All AngularJS applications have a `$rootScope`. The `$rootScope` is the top-most scope and belongs to the DOM element that contains the `ng-app` directive.

When explicit scopes are not set in the application, this is the scope used by AngularJS to bind data and functions. This is why the preceding example works.

To get a better idea of how scopes work, let's attach a controller to a particular DOM element, creating a scope for that element, and then interact with it.

Directives

You saw a few directives in the previous example. In AngularJS a *directive* is a function attached to a DOM element that has the capability to run methods, attach controllers and scope objects, and manipulate the DOM.

When an AngularJS application is launched and Angular starts stepping through the DOM (starting from the DOM element having attribute `ng-app`), it will parse the code collecting and running these directives.

Directives handle all of the hard work of making AngularJS applications dynamic. We've seen a few examples of directives previously, including `ng-model` and `ng-`

bind:

Click here to view code image

```
<body ng-app>
   <p>Name: <input type="text" ng-model="name"></p>
   <span id="output" ng-bind="name"></span>
</body>
```

There are many default directives built into AngularJS, some of which we'll look at next.

Expressions

A double set of curly braces is used to encase an expression directive:

```
{{ expression }}
```

AngularJS expressions are rather like JavaScript expressions, in that they can contain literals, operators, and variables. These are all valid AngularJS expressions:

Click here to view code image

```
{{ 3 + 9 }}

{{ quantity * cost }}

{{ firstName + " " + lastName }}
```

AngularJS expressions are interpreted as data in the exact location where the expression is written.

ng-init

The `ng-init` directive runs at startup, before AngularJS has run any application code. With it you can set default variables prior to running any other functions.

ng-click

The `ng-click` directive attaches a listener to a DOM element. When the element is clicked, AngularJS executes the expression defined in the directive.

ng-repeat

The `ng-repeat` directive iterates through a collection and loads a template for each item. The template it copies is the element having the attribute `ng-repeat`.

```
$scope.departments = [
  { name: 'Sales'},
  { name: 'Support'},
  { name: 'Production'},
  { name: 'Shipping'}
];
```

You can iterate through them using `ng-repeat`:

Click here to view code image

```
<ul>
  <li ng-repeat="dept in departments">{{ dept.name }}</li>
</ul>
```

Here the `<li>` element will be cloned four times to produce the list sent to the view.

Filters

The job of a filter is to select a subset of items from an array and return it as a new array. Here are a few things you might do with an array:

- Format a number to a currency format, using `currency`.
- Select a subset of items from an array, using `filter`.
- Format a string to lowercase, using `lowercase`.
- Order an array by an expression, using `orderBy`.
- Format a string to uppercase, using `uppercase`.

Here's the general syntax for a filter in AngularJS:

Click here to view code image

```
{{ filter_expression | filter : expression : comparator }}
```

Let's suppose you want to apply a currency filter to some numeric data:

Click here to view code image

```
<div ng-app>
Total: <input type="number" ng-model="netTotal">
Tax: <input type="number" ng-model="tax">
<p>Invoice Total = {{ (netTotal + tax) | currency }}</p>
</div>
```

In this example, the expression `{{ netTotal + tax }}` will be evaluated, and the result formatted as currency.

Adding a Filter to a Directive

A filter can also be added to any `ng-` directive by using the pipe character (`|`) followed by a filter description:

Click here to view code image

```
<ul>
   <li ng-repeat="dept in departments | filter: uppercase">{{ dept.name
}}</li>
</ul>
```

This example will display all list entries in uppercase.

Building an AngularJS Application

You now know enough to put together a basic AngularJS application.

Try it Yourself: A Basic AngularJS Application

We'll start with a basic HTML page containing a text input field to accept a user's search string and a `<div>` element to contain a list of search results containing the entered string.

Listing 20.2 shows the basic HTML of the page, with the AngularJS framework already included.

LISTING 20.2 HTML code of the AngularJS Application

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>AngularJS Example</title>
    <script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
</script>
</head>
<body>
    <p>Search Departments: <input type="text"></p>
    <div id="list-container">
       <ul>
           <li></li>
       </ul>
    </div>
</body>
</html>
```

Next, we'll apply the necessary `ng-` directives to the page:

The `ng-app` directive to the `<body>` element, defining this as the container for the AngularJS application.

The `ng-model` directive to the search field, defining it as a model in our MVC framework.

The `ng-repeat` directive to the `<li>` element in our list of search results. The `<li>` element will be repeated once for each search result.

We'll also use the `ng-init` directive to set up some initial data for the application. In a real-world case, this data is more likely to be brought instead from an external source such as a server-side database, but this will serve for our example.

Click here to view code image

```
      ng-init = "departments = [
         { name: 'Sales', contact: 'Marsha Brown'},
         { name: 'Support', contact: 'Dave Price'},
         { name: 'Production', contact: 'Grant Wales'},
         { name: 'Service', contact: 'Sherry Dell'},
         { name: 'Administration', contact: 'Sally Bennett'},
         { name: 'Accounting', contact: 'Kim Sutherland'},
         { name: 'Shipping', contact: 'Sandy Connell'}]"
```

Our initial data comprises an array of fictional departments, each including the department name and the name of the staff contact in charge of it.

Listing 20.3 shows the revised HTML, which also includes a little CSS styling for good measure.

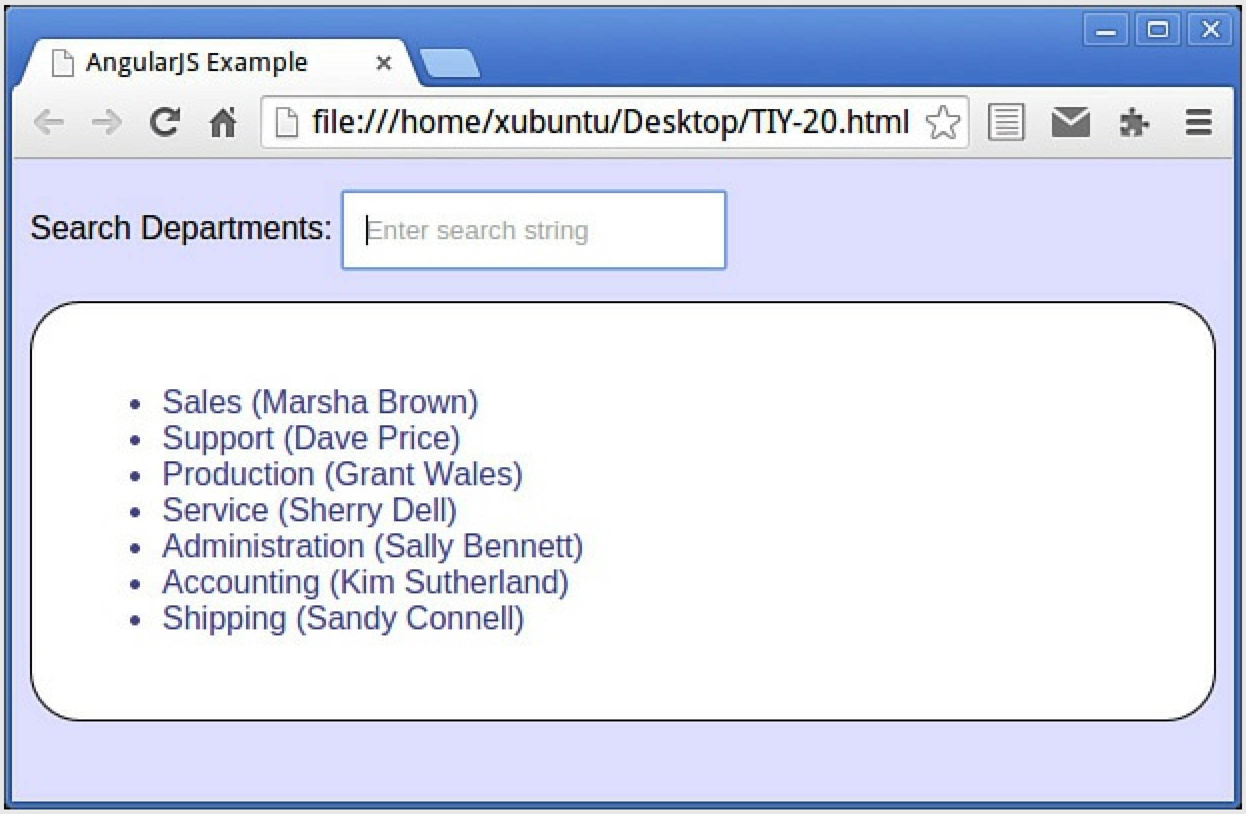
LISTING 20.3 Revised Code of the AngularJS Application

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
<title>AngularJS Example</title>
<style>
     body {
         background-color: #ddf;
         font: 16px bold helvetica, arial, sans-serif;
     }
     input {
         padding: 10px;
     }
     #list-container {
         background-color: white;
         color: #448;
         border-radius: 25px;
         border: 1px solid black;
         padding: 25px;
     }
</style>
<script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js">
</script>
</head>
<body ng-app ng-init = "departments = [
   { name: 'Sales', contact: 'Marsha Brown'},
   { name: 'Support', contact: 'Dave Price'},
   { name: 'Production', contact: 'Grant Wales'},
   { name: 'Service', contact: 'Sherry Dell'},
   { name: 'Administration', contact: 'Sally Bennett'},
   { name: 'Accounting', contact: 'Kim Sutherland'},
   { name: 'Shipping', contact: 'Sandy Connell'}]">
```

```
  <p>Search Departments: <input type="text" placeholder="Enter search
string" ng-model="searchString"></p>
  <div id="list-container">
     <ul>
         <li ng-repeat="dept in departments">{{ dept.name }}</li>
     </ul>
  </div>
</body>
</html>
```

Save this code to an .html file and open it in your browser. You should see the departments and contacts listed in a page looking something like the one in Figure 20.3.



Search Departments: Enter search string

- Sales (Marsha Brown)
- Support (Dave Price)
- Production (Grant Wales)
- Service (Sherry Dell)
- Administration (Sally Bennett)
- Accounting (Kim Sutherland)
- Shipping (Sandy Connell)

FIGURE 20.3 Our AngularJS app ready for use

All well and good, but the search field doesn't currently do anything. We'll fix that by adding a filter to the `ng-repeat` directive, based on the data entered in the search field, as shown in Listing 20.4.

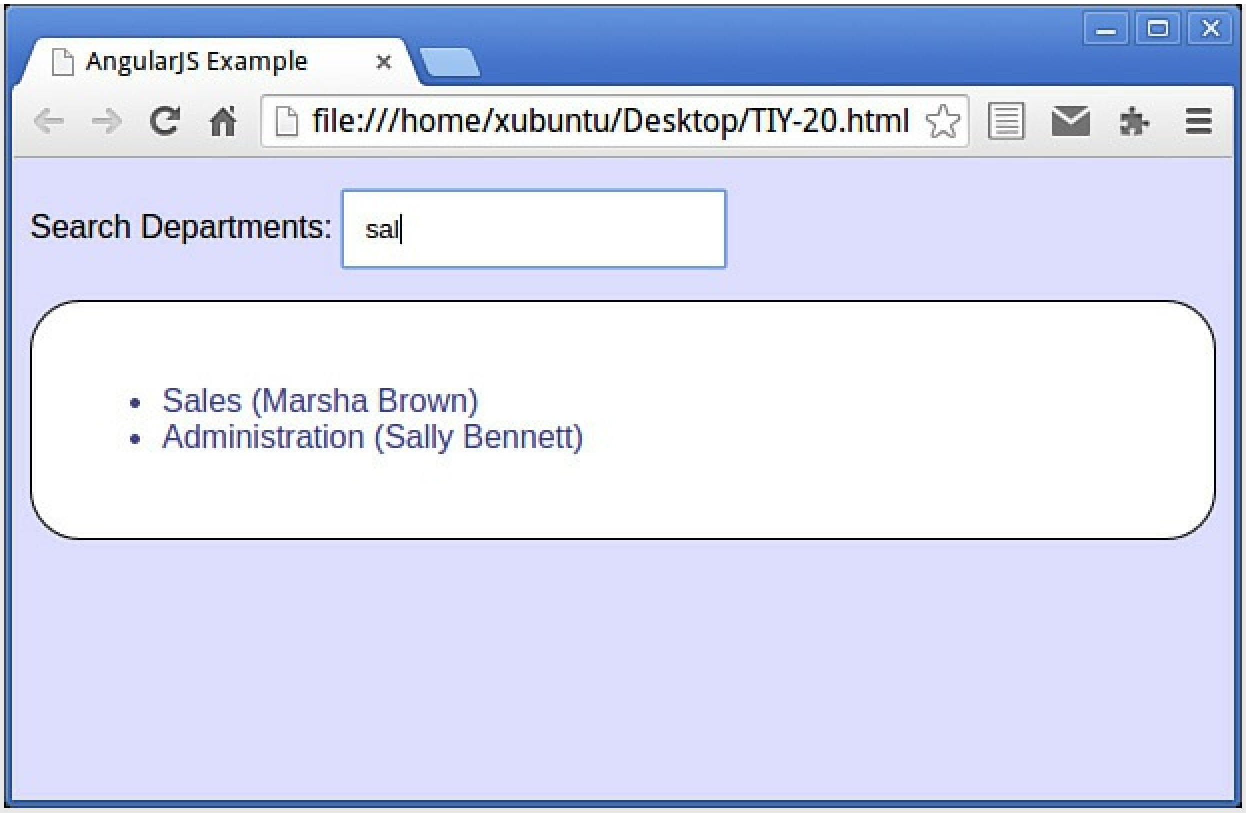
LISTING 20.4 The Finalized AngularJS Application

Click here to view code image

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

```
<title>AngularJS Example</title>
<style>
    body {
        background-color: #ddf;
        font: 16px bold helvetica, arial, sans-serif;
    }
    input {
        padding: 10px;
    }
    #list-container {
        background-color: white;
        color: #448;
        border-radius: 25px;
        border: 1px solid black;
        padding: 25px;
    }
</style>
<script
src="http://ajax.googleapis.com/ajax/libs/angularjs/1.3.14/angular.min.js"
</script>
</head>
<body ng-app>
  <p>Search Departments: <input type="text" placeholder="Enter search
string"
ng-model="searchString"></p>
  <div ng-init = "departments = [
  { name: 'Sales', contact: 'Marsha Brown'},
  { name: 'Support', contact: 'Dave Price'},
  { name: 'Production', contact: 'Grant Wales'},
  { name: 'Service', contact: 'Sherry Dell'},
  { name: 'Administration', contact: 'Sally Bennett'},
  { name: 'Accounting', contact: 'Kim Sutherland'},
  { name: 'Shipping', contact: 'Sandy Connell'}]"></div>
  <div id="list-container">
      <ul>
          <li ng-repeat="dept in departments | filter: searchString">{{
dept.name + " (" + dept.contact + ")" }}</li>
      </ul>
  </div>
</body>
</html>
```

And that’s all we need to do! AngularJS handles the data binding so the filter acts in real time as a user types (see Figure 20.4).



Search Departments: sal

- Sales (Marsha Brown)
- Administration (Sally Bennett)

FIGURE 20.4 The filter directive acts as you type

Summary

In this hour, you learned the basics about the Model-View-Controller framework architecture, and how that can be usefully applied to web applications.

You were introduced to Google's AngularJS framework, and used it to build a simple web application with little or no additional code.

In truth, we've barely touched the surface of what AngularJS can do. Take a look at the official website at https://angularjs.org/ to learn more.

Q&A

Q. What is the background of AngularJS?

A. AngularJS was developed in 2009 by a company called Brat Tech LLC as part of a commercial JSON storage service. It was later released as an open-source library, which Google employees continue to maintain and support.

Q. Where can I get AngularJS documentation and help?

A. The official website at https://angularjs.org/ has links to extensive documentation, tutorials, developer guides, and much more.

Workshop

Try to answer all the questions before reading the subsequent “Answers” section.

Quiz

1. What does the M stand for in an MVC framework?
   a. Mirror
   b. Managed
   c. Model
2. In an AngularJS application, `ng-` directives are added:
   a. To the document head
   b. To page elements
   c. In a separate file linked into the document
3. The `ng-init` directive runs:
   a. On application startup, before the AngularJS application code
   b. When called by the user
   c. After the application terminates

Answers

1. c. Model
2. b. To page elements
3. a. On application startup, before the AngularJS application code

Exercises

- Modify the code of the “Try It Yourself” example to search only within the department name, but still to report both department name and contact name in the displayed list.
- Check out the AngularJS API docs at https://docs.angularjs.org/api and discover just how much more you can do with AngularJS.

Hour 21. JavaScript Beyond the Web Page

What You'll Learn in This Hour:

- Some examples of applications for JavaScript outside straightforward web pages
- How to write a browser extension for Google Chrome

Up to now you've learned a wide range of uses for JavaScript in the writing of web pages. However, JavaScript can also be used for extending browsers by building add-ons and extensions. Also, JavaScript interpreters are embedded in a number of tools apart from web browsers. Such applications often provide their own object model representing the host environment, although the core JavaScript language may remain essentially the same in each instance.

In this hour, you learn about uses for JavaScript above and beyond writing simple web content. You also write your own extension for Google's Chrome browser.

JavaScript Outside the Browser

There are a number of applications for JavaScript to control the actions of other applications in addition to web pages:

- Browser extensions for Google's Chrome, Opera, and Apple's Safari 5 browsers, and widget/gadget collections for Apple's Dashboard, Microsoft, Yahoo!, and Google Desktop can all be written using JavaScript.
- JavaScript is supported in PDF files used by Adobe's Acrobat and Adobe Reader, as well as many third-party applications.
- Adobe tools such as Photoshop, Illustrator, Dreamweaver, and others allow scripting via JavaScript.
- The OpenOffice.org office application suite (and its sibling LibreOffice) have JavaScript as one of the included macro scripting languages. These suites are written largely in Java and provide a JavaScript implementation based on Mozilla Rhino. JavaScript macros can access the application's variables and objects, much like web browsers host scripts that access the browser's Document Object Model (DOM) for a web page.
- Sphere, an open source and cross-platform program for writing role-playing games, and the Unity game engine support JavaScript for scripting.
- Google Apps Script allows users access and control over Google Spreadsheets and other products using JavaScript.
- ActionScript, the programming language used in Adobe Flash, is another

implementation of the ECMAScript standard.

- The Mozilla platform, which is the basis of Firefox, Thunderbird, and other projects, uses JavaScript for the graphical user interface of these applications.

In this hour of the book, you're going to try your hand at one of these—writing an extension for Google's Chrome web browser.

Writing Google Chrome Extensions

Extensions are small applications that run inside a web browser and provide additional services, integrate with third-party websites or data sources, and customize the user's experience of the browser application. A Google Chrome extension is nothing more or less than a collection of files (HTML, CSS, JavaScript, images, and so on) bundled into a .zip file (although it's renamed as a .crx file).

The extension basically creates a web page that can use all the interface elements that the browser provides to regular web pages, including JavaScript libraries, CSS style sheets, `XMLHttpRequest` objects, and so on.

Extensions can interact with web pages or servers, and can also interact via program code with browser features such as bookmarks and tabs.

Building a Simple Extension

The first step is to create a folder on your computer to contain the code for your extension.

Each extension has a manifest file, named manifest.json, that is formatted in JSON and provides important information.

The manifest file can contain a wide range of parameters and options, but here we'll begin with a basic example. In your new folder create a text file called manifest.json and edit it like this:

Click here to view code image

```
{
    "name": "My First Extension",
    "version": "1.0",
    "manifest_version": 2,
    "description": "Hello World extension.",
    "browser_action": {
        "default_icon": "icon.png",
        "default_popup": "popup.html"
    },
    "web_accessible_resources": [
        "icon.png",
        "popup.js"
    ]
}
```

Put an icon called icon.png in the same folder—I used a small graphic image of a star, but you can use whatever you want. Create the file popup.html listed in Listing 21.1 and put that in the folder too.

LISTING 21.1 popup.html Google Chrome Extension

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <style>
        body {
            width: 350px;
        }
        div {
            border: 1px solid black;
            padding: 20px;
            font: 20px normal helvetica, verdana, sans-serif;
        }
    </style>
    <script src="popup.js"></script>
</head>
<body>
</body>
</html>
```

Here is the JavaScript code contained in popup.js:

Click here to view code image

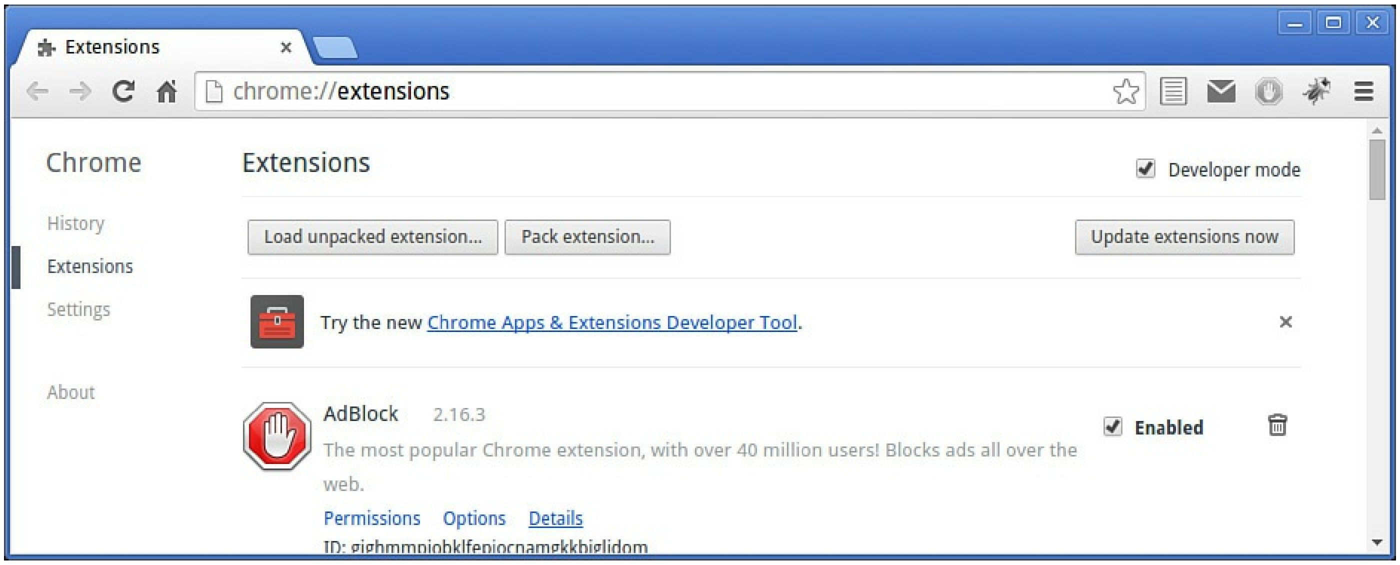
```
function sayHello() {
    var message = document.createTextNode("Hello World!");
    var out = document.createElement("div");
    out.appendChild(message);
    document.body.appendChild(out);
}
window.onload = sayHello;
```

All this does is, on page load, create a `<div>` element containing the message "Hello World!" and append it to the DOM's `<body>` element.

Now display Chrome's Extensions page by clicking the settings icon and selecting **More Tools > Extensions**.

Click the box next to Developer Mode to show a little more information.

Then click the Load Unpacked Extensions button. Navigate to the folder containing your extension and select it. You should see something like Figure 21.1.


Extensions

chrome://extensions

Chrome

History

Extensions

Settings

About

Extensions

Developer mode

Load unpacked extension... Pack extension... Update extensions now

Try the new Chrome Apps & Extensions Developer Tool.

AdBlock 2.16.3

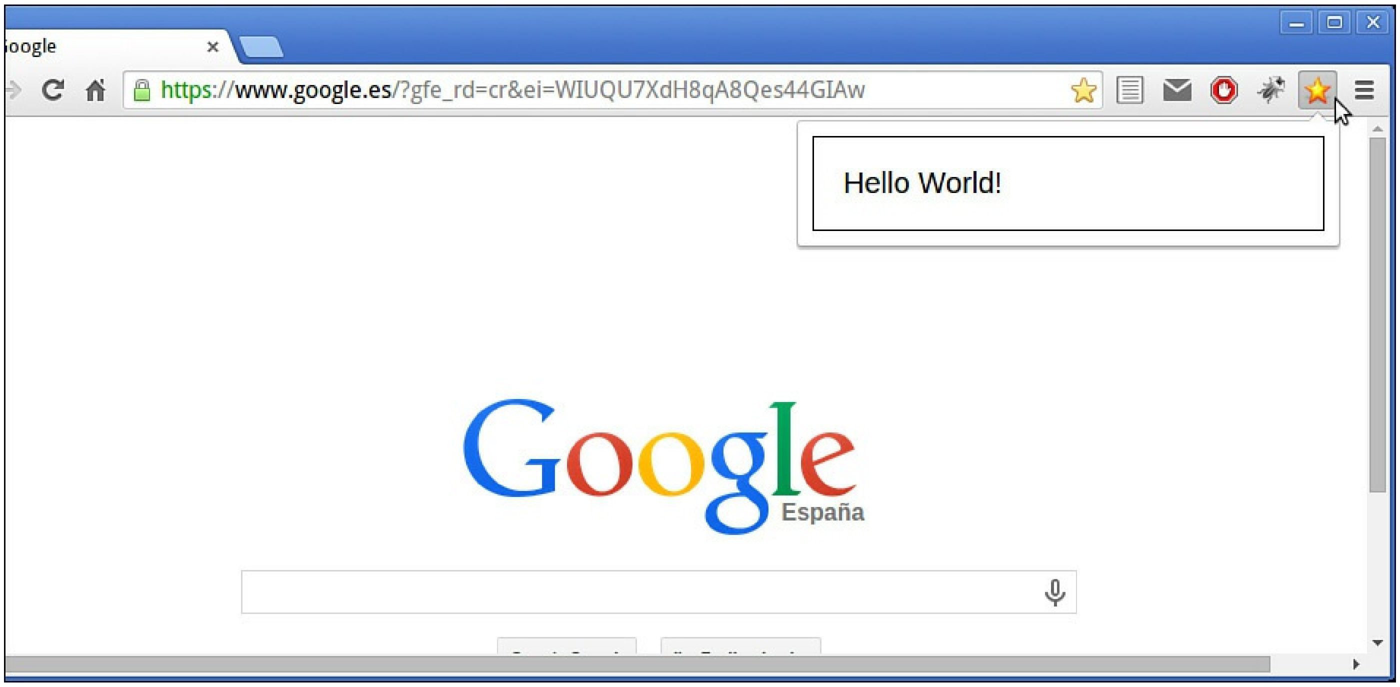
The most popular Chrome extension, with over 40 million users! Blocks ads all over the web.

Permissions Options Details

Enabled

FIGURE 21.1 Your new extension visible on the Extensions page

Make sure the extension is enabled by checking the box next to it. You can now run your extension by clicking on the toolbar icon, as shown in Figure 21.2.

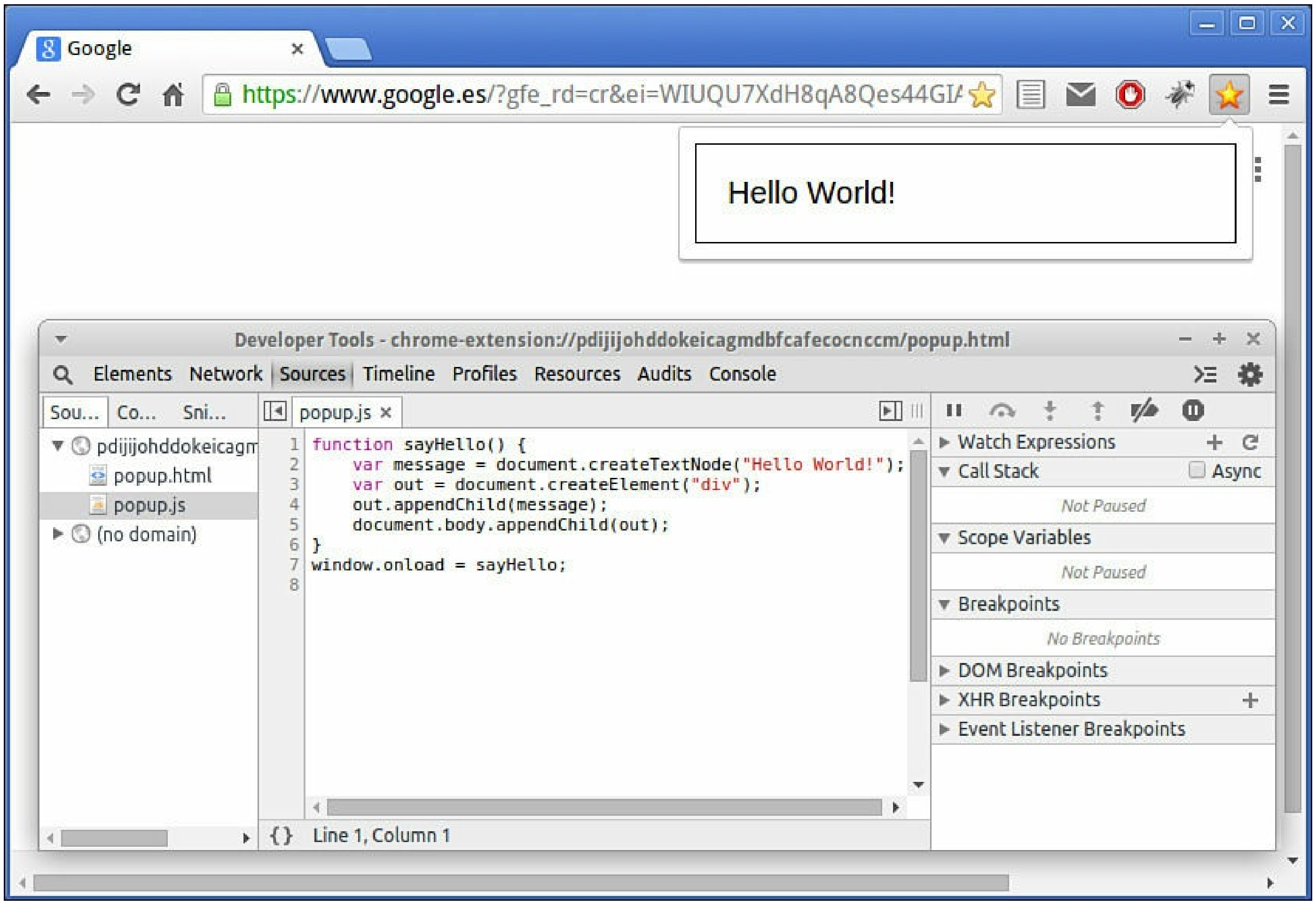

https://www.google.es/?gfe_rd=cr&ei=WIUQU7XdH8qA8Qes44GIAw

Hello World!

FIGURE 21.2 Hello World as a Google Chrome extension

Debugging the Extension

Right-click on the icon that launches your extension, and you see a content menu containing options to enable and disable the extension, plus an option called *Inspect popup*. Click on that and Chrome's Developer Tools pop open to let you examine the pop-up window, as shown in Figure 21.3.



```
function sayHello() {
    var message = document.createTextNode("Hello World!");
    var out = document.createElement("div");
    out.appendChild(message);
    document.body.appendChild(out);
}
window.onload = sayHello;
```

FIGURE 21.3 Inspecting the pop-up window

Try it Yourself: A Chrome Extension to Get Airport Information

This time you're going to make a Chrome extension that's a little more useful. With the help of the jQuery library, your pop-up is going to retrieve current information about U.S. airports.

Tip

Refer to Hour 15, "A Closer Look at jQuery," if you need a refresher on the jQuery library.

To do that, you're going to have your code make an Ajax call to an information feed at http://services.faa.gov/. To demonstrate how this service works, open your browser and navigate to http://services.faa.gov/airport/status/SFO?format=application/json.

"SFO" is the three-letter code for San Francisco International airport; you can replace it in the preceding URL with the code for another U.S. airport; for example, you could use LAX for Los Angeles International or SEA for Seattle-

Tacoma International.

Tip

You can read the airport codes and see their locations at http://www.fly.faa.gov/flyfaa/usmap.jsp.

The format parameter tells the service that you want the information returned as a JSON string:

Click here to view code image

```
{"name":"San Francisco
International","ICAO":"KSFO","state":"California","status":
{"avgDelay":"",
"closureEnd":"", "closureBegin":"","type":"","minDelay":"","trend":"",
"reason":"No known delays for this
airport.","maxDelay":"","endTime":""},
"delay":"false","IATA":"SFO","city":"San Francisco","weather":
{"weather":"Partly
Cloudy",
"meta":{"credit":"NOAA's National Weather
Service","url":"http://weather.gov/",
"updated":"1:56 AM Local"},"wind":"Southwest at 9.2mph","temp":"44.0 F
(6.7 C)",
"visibility":"10.00"}}
```

Your code will parse this returned information and use it to construct a more user-friendly display.

To begin the project, create a new directory somewhere on your computer and call it “airport.” In this directory, you need three files, as in the previous example.

An Icon File

Choose an icon to display on your Chrome toolbar and from which to launch the extension. I used a 20 × 20 pixel airplane icon in a file called plane.png, but you can use any icon you have on hand.

The **manifest.json** File

The manifest file will be pretty familiar from the previous example, but with one notable addition: a new parameter, `permissions`. You are going to make an Ajax call to services.faa.gov to retrieve the information you want, and Ajax calls can only be made to pages on the same domain as the caller; adding a `permissions` entry allows Chrome to fulfil this requirement by sending a suitable header to the server. The manifest.json file is shown in Listing 21.2.

LISTING 21.2 The manifest.json File

Click here to view code image

```
{
    "name": "Airport Information",
    "version": "1.0",
    "manifest_version": 2,
    "description": "Information on US airports",
    "browser_action": {
        "default_icon": "plane.png",
        "default_popup": "popup.html"
    },
    "web_accessible_resources": [
    "plane.png",
    "popup.js"
  ],
    "permissions": [
    "http://services.faa.gov/"
  ]
}
```

The HTML File

Once again the main HTML file will be called popup.html. You can call it something else if you want to, so long as you edit manifest.json and suitably set the value of the “popup” parameter.

The simple HTML page is shown listed in Listing 21.3.

LISTING 21.3 The Basic HTML File popup.html

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Airport Information</title>
    <style>
        body {
            width:350px;
            font: 12px normal arial, verdana, sans-serif;
        }
        #info {
            border: 1px solid black;
            padding: 10px;
        }
    </style>
</head>
<body>
```

```
    <h2>Airport Information</h2>
    <input type=Text id="airportCode" value="SFO" size="6" />
    <input id="btn" type="button" value="Get Information" />
    <div id="info"></div>
</body>
</html>
```

Apart from a little CSS styling, the page only contains a few items: an input field to accept the airport code, with default value set to SFO, a button to request that data is fetched, and a `<div>` to hold the returned results.

Now you need to start adding JavaScript to the page.

You're going to use jQuery to simplify things, so first you need to include that. The Google Chrome security policy doesn't allow the use of a content delivery network, so we need to download and include a copy of the jQuery library:

Click here to view code image

```
<script src="jquery-1.11.2.min.js" /></script>
```

When the page has fully loaded, you need to attach code to the Get Information button. The button needs to assemble the required URL based on the airport code value entered in the input field and instigate the Ajax call. Since the remote service may take some moments to respond, it would also be good if the user received a little message to indicate that the program was working.

Here's the code to carry out these tasks:

Click here to view code image

```
$(document).ready(function(){
   $("#btn").click(function(){
      $("#info").html("Getting information ...");
      var code = $("#airportCode").val();
      $.get("http://services.faa.gov/airport/status/" + code + "?
format=application/json",
         '',
         function(data){
            displayData(data);
         }
      );
   });
});
```

Once the page has loaded, jQuery adds code to the `onclick` event handler of the button.

First it uses jQuery's `html()` method to add a user message to the output `<div>` element. This message will later be overwritten when the "real" information is received.

Click here to view code image

```
$("#info").html("Getting information ...");
```

Next, the desired airport code is retrieved from the input field:

Click here to view code image

```
var code = $("#airportCode").val();
```

Then the Ajax call is assembled, here using `GET`:

Click here to view code image

```
    $.get("http://services.faa.gov/airport/status/" + code + "?
    format=application/
    json",
        '',
        function(data){
            displayData(data);
        }
    );
```

The callback function specified for the Ajax call is `displayData()`, which will format the returned data and display it to the user. Here's the complete contents of popup.js. including the callback function:

Click here to view code image

```
    function displayData(data) {
       var message = "Airport: " + data.name + "<br />";
       message += "<h3>STATUS:</h3>";
       for (i in data.status) {
          if(data.status[i] != "") message += i + ": " + data.status[i] + "
    <br />";
       }
       message += "<h3>WEATHER:</h3>";
       for (i in data.weather) {
          if(i != "meta") message += i + ": " + data.weather[i] + "<br />";
       }
       $("#info").html(message);
    }
    $(document).ready(function(){
       $("#btn").click(function(){
          $("#info").html("Getting information ...");
          var code = $("#airportCode").val();
          $.get("http://services.faa.gov/airport/status/" + code + "?
    format=application/json",
             '',
             function(data){
                displayData(data);
             }
          );
       });
    });
```

Recall from Hour 10, "Meet JSON," that JSON data can be interpreted directly as a hierarchy of JavaScript objects. The `displayData(data)` function takes the returned JSON object data and picks out `data.name` (a string), `data.status`, and `data.weather` (themselves objects) from which to construct the message.

Tip

Look back a few pages to the JSON data returned from the remote server to see how these values were encoded.

The complete HTML page with code included is in Listing 21.4.

LISTING 21.4 The Complete popup.html for the Extension

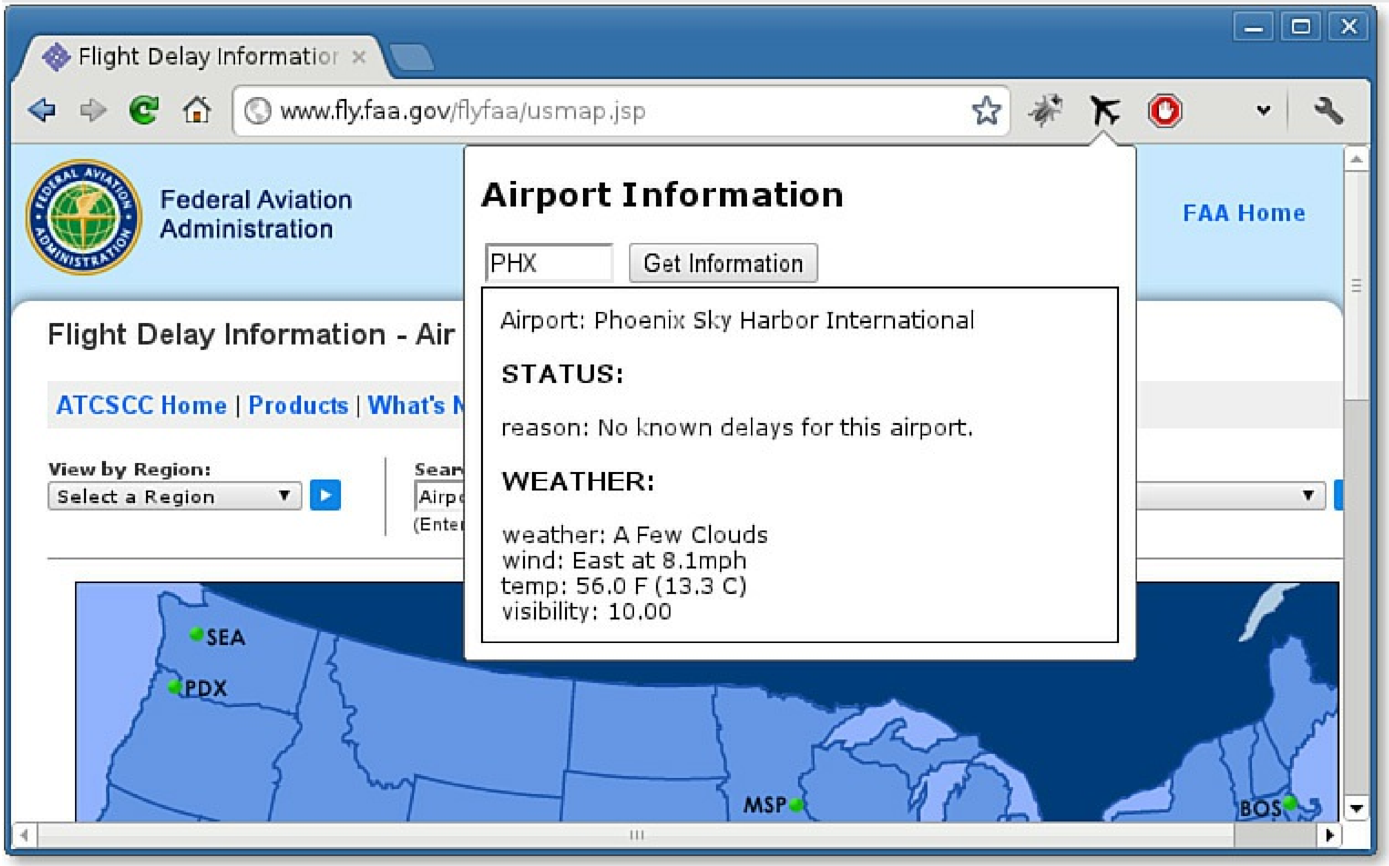
Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Airport Information</title>
    <style>
        body {
            width:350px;
            font: 12px normal arial, verdana, sans-serif;
        }
        #info {
            border: 1px solid black;
            padding: 10px;
        }
    </style>
    <script src="jquery-1.11.2.min.js" /></script>
    <script src="popup.js"></script>
</head>
<body>
    <h2>Airport Information</h2>
    <input type=Text id="airportCode" value="SFO" size="6" />
    <input id="btn" type="button" value="Get Information" />
    <div id="info"></div>
</body>
</html>
```

Having assembled the required files in their allocated directory, you can add the extension to Google Chrome exactly as in the previous example.

Clicking on the associated icon brings up a small form where you can enter the airport code of your choice. Clicking the Get Information button will cause the program to consult the remote service, assemble the returned information into a readable form, and present it in the pop-up window.

Figure 21.4 shows the extension in operation.



Airport Information

PHX Get Information

Airport: Phoenix Sky Harbor International

STATUS:

reason: No known delays for this airport.

WEATHER:

weather: A Few Clouds
wind: East at 8.1mph
temp: 56.0 F (13.3 C)
visibility: 10.00

Flight Delay Informatio
www.fly.faa.gov/flyfaa/usmap.jsp
Federal Aviation Administration
FAA Home
Flight Delay Information - Air
ATCSCC Home | Products | What's N
View by Region:
Select a Region
SEA
PDX
MSP
BOS

FIGURE 21.4 The Airport Information extension

Packaging the Extension

When you've finished developing your extension, click on the Pack Extension button in the Extensions page. Your extension will be packed into a .crx file for you. You can serve the .crx file from your web pages, and your visitors will be able to install it on their own copy of Google Chrome.

Going Further

The exercises of this hour barely scratched the surface of what can be done with Chrome extensions. Because Chrome has good support for HTML5 and CSS3, you can use the latest web technologies such as `canvas`, `localStorage`, and CSS animations in your extensions, as well as access to external APIs and data sources. Your extensions can even add buttons to the Chrome browser's user interface, or create pop-up notifications that exist outside the browser window.

Summary

In this hour, you learned about some applications of JavaScript beyond its use in HTML web pages. As an example, you wrote a small extension for Google's Chrome browser using JavaScript.

Q&A

Q. Can I write a Firefox extension in a similar way to the Chrome extension described here?

A. The Mozilla way of creating extensions is a little more complex; in addition to JavaScript, you'll have to mess a little with XML too. You'll find some good information to help you get started at https://developer.mozilla.org/en/XUL_School/Getting_Started_with_Firefox_Exte

Q. Is it possible to write whole applications in JavaScript that don't have to run inside a browser?

A. Yes it is. As an example, take a look at Node.js (http://www.nodejs.org). Node.js is a platform built on top of Google Chrome's JavaScript runtime engine and designed for building server-side network applications such as web servers, chat applications, network monitoring tools, and much more.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. Information about a Google Chrome extension is contained in a file called:

a. manifest.json

b. manifest.js

c. manifest.txt

2. A Google Chrome extension is distributed as which type of file?

a. .js

b. .xml

c. .crx

Answers

1. a. manifest.json

2. c. Google Chrome extensions can be distributed as a .crx file.

Exercises

- Browse the available JSON APIs listed at http://www.programmableweb.com/apitag/weather?format=JSON and try writing your own simple Chrome extension to display the data.

- Take a look at the documentation for Node js (http://www.nodejs.org) to see how JavaScript can be used to write server-side scripts.

Part VII: Learning the Trade

Hour 22. Good Coding Practice

What You'll Learn in This Hour:

- How to avoid overuse of JavaScript
- Writing readable and maintainable code
- About graceful degradation
- About progressive enhancement
- How to separate style, content, and code
- Writing unobtrusive JavaScript
- Using feature detection
- Avoiding inline code such as event handlers
- How to handle errors well

JavaScript has gained an unfortunate reputation in certain circles. Since its main goal as a scripting language was to add functionality to web page designs, accessibility for first-time programmers has always been an important aspect of the language. Unfortunately, that has often led to poorly written code being allowed into web pages, leading to frustration for more software-savvy users.

Throughout the book so far I've made reference to aspects of coding that are good and bad. In this hour we pull all that together to form some general guidelines for good coding practice.

Don't Overuse JavaScript

How much JavaScript do you need? There's often a temptation to include JavaScript code and enhanced interaction where it's not strictly necessary or advisable.

- It's important to remember that your users are likely to spend most of their Internet time on sites other than yours. Experienced Internet users become accustomed to popular interface components such as menus, breadcrumb trails, and tabbed browsing. These elements are popular, in general, because they work well, can be made to look good, and don't require the user to read a manual first. Is familiarity with a site's operation likely to increase a user's productivity more than the potential benefits of your all-new whizz-bang design?
- Many of the visual effects that once needed to be coded in JavaScript can now be achieved perfectly well using CSS. Where both approaches are possible (image rollovers and some types of menus come immediately to mind), CSS is usually

preferable. It’s well supported across browsers (despite a few variations) and isn’t as commonly turned off by the user. In the rare case that CSS isn’t supported, the page is rendered as standard HTML, usually leaving a page that’s at least perfectly functional, even if it’s not so pretty.

- Users in many areas of the world are still using outdated, underpowered, hand-me-down computers and may also have slow and/or unreliable Internet access. The CPU cycles taken up by your unnecessary code may be precious to them.
- In some cases you may cost yourself a degree of search engine page rank, since their spiders don’t always correctly index content that’s been generated by JavaScript, or designs that require it for navigation.

Used carefully and with forethought, JavaScript can be a great tool, but sometimes you can have too much of a good thing.

Writing Readable and Maintainable Code

There is no way of knowing who will one day need to read and understand your code. Even if that person is you, several years and many projects may have intervened; the code that is so familiar to you at the time of writing can seem mystifying further down the line. If somebody else has to interpret your code, they may not share your coding style, naming conventions, or areas of expertise, and you may not be available to help them out.

Use Comments Sensibly

Well-chosen comments at critical places in your code can make all the difference in such situations. Comments are your notes and pointers for those who come later. The trick is in deciding what comments are likely to be helpful. The subject has often raised debate, and opinions vary widely, so what follows is largely my own opinion.

It’s perhaps reasonable to assume that the person who ends up reading your code has an understanding of JavaScript, so a commentary on the way the language itself works is going too far; JavaScript developers may vary widely in their styles and abilities, but the one thing we do all share is the language syntax!

Harder to interpret when reading code are the thought processes and algorithms that lie behind the code’s operation. Personally, when reading code written by others I find it helpful to see

- A prologue to any object or function containing more than a few lines of simple code.

[Click here to view code image](#)

```
function calculateGroundAngle(x1, y1, z1, x2, y2, z2) {
    /**
```

```
        * Calculates the angle in radians at which
        * a line between two points intersects the
        * ground plane.
        * @author Phil Ballard phil@www.example.com
        */
        if(x1 > 0) {
            .... more statements
```

- Inline comments wherever the code would otherwise be confusing or prone to misinterpretation.

Click here to view code image

```
    // need to use our custom sort method for performance reasons
    var finalArray = rapidSort(allNodes, byAngle) {
        .... more statements
```

- A comment wherever the original author can pass on specialist knowledge that the reader is unlikely to know.

Click here to view code image

```
    // workaround for image onload bug in browser X version Y
    if(!loaded(image1)) {
        .... more statements
```

- Instructions for commonly used code modifications.

Click here to view code image

```
    // You can change the following dimensions to your preference:
    var height = 400px;
    var width = 600px;
```

Tip

Various schemes use code comments to help you generate documentation for your software. See, for example, http://code.google.com/p/jsdoc-toolkit/.

Choose Helpful File, Property, and Method Names

The amount of comments required in your source code can be greatly reduced by making the code as self-commenting as possible. You can go some way toward this by choosing meaningful human-readable names for methods and properties.

JavaScript has rules about the characters allowed in the names of methods (or functions) and properties (or variables), but there's still plenty of scope to be creative and concise.

A popular convention is to put the names of constants into all uppercase:

```
MONTHS_PER_YEAR = 12;
```

For regular function, method, and variable names, so-called CamelCase is a popular option; names constructed from multiple words are concatenated with each word except the first initialized. The first letter can be upper- or lowercase:

```
    var memberSurname = "Smith";
    var lastGroupProcessed = 16;
```

It's recommended that constructor functions for instantiating objects have the first character capitalized:

Click here to view code image

```
    function Car(make, model, color) {
        .... statements
    }
```

The capitalization provides a reminder that the `new` keyword needs to be used:

Click here to view code image

```
  var herbie = new Car('VW', 'Beetle', 'white');
```

Reuse Code Where You Can

Generally, the more you can modularize your code, the better. Take a look at this function:

Click here to view code image

```
    function getElementArea() {
        var high = document.getElementById("id1").style.height;
        var wide = document.getElementById("id1").style.width;
        return high * wide;
    }
```

The function attempts to return the area of screen covered by a particular HTML element. Unfortunately it can only ever work with an element having `id = "id1"`, which is really not very helpful at all.

Collecting your code into modules such as functions and objects that you can use and reuse throughout your code is a process known as *abstraction*. We can give the function *a higher level of abstraction* to make its use more general by passing as an argument the ID of the element to which the operation should be applied:

Click here to view code image

```
    function getElementArea(elementId) {
        var elem = document.getElementById(elementId);
        var high = elem.style.height;
        var wide = elem.style.width;
        return parseInt(high) * parseInt(wide);
    }
```

You could now call your function into action for any element having an ID:

Click here to view code image

```
    var area1 = getElementArea("id1");
```

```
var area2 = getElementArea("id2");
```

Don’t Assume

What happens in the previous function when we pass a value for `elementId` that doesn’t correspond to any element on the page? The function causes an error, and code execution halts.

The error is to assume that an allowable value for `elementId` will be passed. Let’s edit the function `getElementArea()` to carry out a check that the page element does indeed exist, and also that it has a numeric area:

Click here to view code image

```
function getElementArea(elementId) {
    if(document.getElementById(elementId)) {
        var elem = document.getElementById(elementId);
        var high = elem.style.height;
        var wide = elem.style.width;
        var area = parseInt(high) * parseInt(wide);
        if(!isNaN(area)) {
            return area;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

That’s an improvement. Now the function will return `false` if it cannot return a numeric area, either because the relevant page element couldn’t be found, or because the ID corresponded to a page element without accessible `width` and `height` properties.

Graceful Degradation

Among the earliest web browsers were some that didn’t even support the inclusion of images in HTML. When the `<img>` element was introduced, a way was needed to allow those text-only browsers to present something helpful to the user whenever such a nonsupported tag was encountered.

In the case of the `<img>` tag, that facility was provided by the `alt` (alternative text) attribute. Web designers could assign a string of text to `alt`, and text-only browsers would display this text to the user instead of showing the image. At the whim of the page designer, the `alt` text might be simply a title for the image, a description of what the picture would have displayed, or a suggestion for an alternative source of the information that would have been carried in the graphic.

This was an early example of *graceful degradation*, the process by which a user whose browser lacks the required technical features to make full use of a web page’s design—

or has those features disabled—can still benefit as fully as possible from the site's content.

Let's take JavaScript itself as another example. Virtually every browser supports JavaScript, and few users turn it off. So do you really need to worry about visitors who don't have JavaScript enabled? The answer is probably yes. One type of frequent visitor to your site will no doubt be the spider program from one of the search engines, busy indexing the pages of the Web. The spider will attempt to follow all the navigation links on your pages to build a full index of your site's content; if such navigation requires the services of JavaScript, you may find some parts of your site not being indexed. Your search ranking will probably suffer as a result.

Another important example lies in the realm of accessibility. No matter how capable a browser program is, there are some users who suffer with other limitations, such as perhaps the inability to use a mouse, or the necessity to use screen-reading software. If your site does not cater to these users, they're unlikely to return.

Progressive Enhancement

When we talk about graceful degradation, it's easy to imagine a fully functional web page with all the bells and whistles providing charitable assistance to users whose browsers have lesser capabilities.

Supporters of *progressive enhancement* tend to look at the problem from the opposite direction. They favor the building of a stable, accessible, and fully functional website, the content of which can be accessed by just about any imaginable user and browser, to which they can later add extra layers of additional usability for those who can take advantage of them.

This ensures that the site will work for even the most basic browser setup, with more advanced browsers simply gaining some additional enhancements.

Separate Style, Content, and Code

The key resource of a web page employing progressive enhancement techniques is the content. HTML provides markup facilities to allow you to describe your content semantically; the markup tags themselves identify page elements as being headings, tables, paragraphs, and so on. We might refer to this as the *semantic layer*.

What this semantic layer should ideally *not* contain is any information about how the page should appear. You can add this additional information afterwards, using CSS techniques to form the *presentation layer*. By linking external CSS stylesheets into the document, you avoid any appearance-related information from appearing in the HTML markup itself. Even a browser having no understanding of CSS, however, can still access and display all of the page's information, even though it might not look so pretty.

When you now come to add JavaScript into the mix, you do so as yet another notional layer—you might think of it as the *behavior layer*. Users without JavaScript still have access to the page content via the semantic markup; if their browser understands CSS, they'll also benefit from the enhanced appearance of the presentation layer. If the JavaScript of the behavior layer is applied correctly, it will offer more functionality to those who can use it, without prejudicing the abilities of the preceding layers.

To achieve that, you need to write JavaScript that is *unobtrusive*.

Unobtrusive JavaScript

There is no formal definition of unobtrusive JavaScript, but the concepts upon which it's built all involve maintaining the separation between the behavior layer and the content and presentation layers.

Leave That HTML Alone

The first and perhaps most important consideration is the removal of JavaScript code from the page markup. Early applications of JavaScript clutter the HTML with inline event handlers such as the `onClick` event handler in this example:

Click here to view code image

```
<input type="button" style="border: 1px solid blue;color: white"
onclick="doSomething()" />
```

Inline style attributes, such as the one in the preceding example, can make the situation even worse.

Thankfully you can effectively remove the style information to the style layer, for example, by adding a `class` attribute to the HTML tag referring to an associated style declaration in an external CSS file:

Click here to view code image

```
<input type="button" class="blueButtons" onclick="doSomething()" />
```

And in the associated CSS definitions:

```
.blueButtons {
    border: 1px solid blue;
    color: white;
}
```

Tip

You could, of course, define your style rule for the button via any one of a number of different selectors, including the `input` element or via an `id` instead of a `class` attribute.

To make your JavaScript unobtrusive you can employ a similar technique to the one we just used for CSS. By adding an `id` attribute to a page element within the HTML markup, you can attach the required `onClick` event listener from within your external JavaScript code, keeping it out of the HTML markup altogether. Here's the revised HTML element:

Click here to view code image

```
<input type="button" class="blueButtons" id="btn1" />
```

The `onClick` event handler is attached from within your JavaScript code:

Click here to view code image

```
function doSomething() {
    .... statements ....
}
document.getElementById("btn1").onclick = doSomething;
```

Caution

Remember that you can't use DOM methods until the DOM is available, so any such code must be attached via a method such as `window.onload` to guarantee DOM availability. There are plenty of examples throughout this book.

Use JavaScript Only as an Enhancement

In the spirit of progressive enhancement, you want your page to work even if JavaScript is turned off. Any improvements in the usability of the page that JavaScript may add should be seen as a bonus for those users whose browser setup permits them.

Let's imagine you want to write some form validation code—a popular use for JavaScript. Here's a little HTML search form:

Click here to view code image

```
<form action="process.php">
<input id="searchTerm" name="term" type="text" /><br />
<input type="button" id="btn1" value="Search" />
</form>
```

You want to write a routine to prevent the form from being submitted if the search field is blank. You might write this function `checkform()`, which will be attached to the `onClick` handler of the search button:

Click here to view code image

```
function checkform() {
    if(document.forms[0].term.value == "") {
        alert("Please enter a search term.");
        return false;
    } else {
        document.forms[0].submit();
    }
}
window.onload = function() {
    document.getElementById("btn1").onclick = checkform;
}
```

That should work just fine. But what happens when JavaScript is switched off? The button now does nothing at all, and the form can't be submitted by the user. Your users would surely prefer that the form could be used, albeit without the *enhancement* of input checking.

Let's change the form slightly to use an input button of `type="submit"` rather than `type="button"`, and edit the `checkform()` function:

Click here to view code image

```
    <form action="process.php">
        <input id="searchTerm" name="term" type="text" /><br />
        <input type="submit" id="btn1" value="Search" />
    </form>
```

Here's the modified `checkform()` function:

Click here to view code image

```
    function checkform() {
        if(document.forms[0].term.value == "") {
            alert("Please enter a search term.");
            return false;
        } else {
            return true;
        }
    }
    window.onload = function() {
        document.getElementById("btn1").onclick = checkform;
    }
```

If JavaScript is active, returning a value of `false` to the submit button will prevent the default operation of the button, preventing form submission. Without JavaScript, however, the form will still submit when the button is clicked.

Feature Detection

Where possible, try to directly detect the presence or absence of browser features, and have your code use those features only where available.

As an example, let's look at the `clipboardData` object, which at the time of writing is only supported in Internet Explorer. Before using this object in your code, it's a good idea to perform a couple of tests:

- Does JavaScript recognize the object's existence?
- If so, does the object support the method you want to use?

The following function `setClipboard()` attempts to write a particular piece of text directly to the clipboard using the `clipboardData` object:

Click here to view code image

```
    function setClipboard(myText){
        if((typeof clipboardData != 'undefined') &&
    (clipboardData.setData)){
            clipboardData.setData("text", myText);
        } else {
            document.getElementById("copytext").innerHTML = myText;
            alert("Please copy the text from the 'Copy Text' field to your
    clipboard");
        }
    }
```

First it tests for the object's existence using `typeof`:

Click here to view code image

```
 if((typeof clipboardData != 'undefined') ....
```

Note

The `typeof` operator returns one of the following, depending on the type of the operand:

`"undefined"`, `"object"`, `"function"`, `"boolean"`, `"string"`, or `"number"`

Additionally, the function insists that the `setData()` method must be available:

Click here to view code image

```
 ... && (clipboardData.setData)){
```

If either test fails, the user is offered an alternative, if less elegant, method of getting the text to the clipboard; it is written to a page element and the user is invited to copy it:

Click here to view code image

```
    document.getElementById("copytext").innerHTML = myText;
    alert("Please copy the text from the 'copytext' field to your
    clipboard");
```

At no point does the code try to explicitly detect that the user's browser is Internet Explorer (or any other browser); should some other browser one day implement this functionality, the code should detect it correctly.

Handling Errors Well

When your JavaScript program encounters an error of some sort, a warning or error will be created inside the JavaScript interpreter. Whether and how this is displayed to the user depends on the browser in use and the user's settings; the user may see some form of error message, or the failed program may simply remain silent but inactive.

Neither situation is good for the user; he or she is likely to have no idea what has gone wrong, or what to do about it.

As you try to write your code to handle a wide range of browsers and circumstances, it's possible to foresee some areas in which errors might be generated. Examples include

- The uncertainty over whether a browser fully supports a certain object, and whether that support is standards compliant
- Whether an independent procedure has yet completed its execution, such as an external file being loaded

Using **try** and **catch**

A useful way to try to intercept potential errors and deal with them cleanly is by using the `try` and `catch` statements.

The `try` statement allows you to attempt to run a piece of code. If the code runs without errors, all is well; however, should an error occur you can use the `catch` statement to intervene before an error message is sent to the user, and determine what the program should then do about the error.

```
try {
    doSomething();
}
catch(err) {
    doSomethingElse();
}
```

Note the syntax:

```
catch(identifier)
```

Here `identifier` is an object created when an error is caught. It contains information about the error; for instance, if you wanted to alert the user to the nature of a JavaScript runtime error, you could use a code construct like

```
catch(err) {
    alert(err.description);
}
```

to open a dialog containing details of the error.

Try it Yourself: Converting Code into Unobtrusive Code

From time to time you may find yourself in the position of having to modernize code to make it less obtrusive. Let's do that with some code we wrote way back in Hour 4, “DOM Objects and Built-in Objects,” presented once again here in Listing 22.1.

LISTING 22.1 An Obtrusive Script

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Current Date and Time</title>
    <style>
        p {font: 14px normal arial, verdana, helvetica;}
    </style>
    <script>
        function telltime() {
            var out = "";
            var now = new Date();
            out += "<br />Date: " + now.getDate();
            out += "<br />Month: " + now.getMonth();
            out += "<br />Year: " + now.getFullYear();
            out += "<br />Hours: " + now.getHours();
            out += "<br />Minutes: " + now.getMinutes();
            out += "<br />Seconds: " + now.getSeconds();
            document.getElementById("div1").innerHTML = out;
        }
    </script>
</head>
<body>
    The current date and time are:<br/>
    <div id="div1"></div>
    <script>
        telltime();
    </script>
    <input type="button" onclick="location.reload()" value="Refresh" />
</body>
</html>
```

As it stands, this script has a number of areas of potential improvement:

The JavaScript statements are placed between `<script>` and `</script>` tags on the page; they would be better in a separate file.

The button has an inline event handler.

A user without JavaScript would simply see a page with a nonfunctioning button.

First, let’s move all the JavaScript to a separate file and remove the inline event handler. We also give the button an id value, so we can identify it in JavaScript to add the required event handler via our code.

Next, we need to address the issue of users without JavaScript enabled. We use the `<noscript>` page element so that users without JavaScript enabled will see, instead of the button, a short message with a link to an alternative source of time information:

Click here to view code image

```
<noscript>
    Your browser does not support JavaScript<br />
    Please consult your computer's operating system for local date and
time information or click <a href="clock.php" target="_blank">HERE</a>
to read the server time.
</noscript>
```

Tip

The `<noscript>` element provides additional page content for users with disabled scripts or with a browser that can't support client-side scripting. Any of the elements that you can put in the `<body>` element of an HTML page can go inside the `<noscript>` element, and will automatically be displayed if scripts cannot be run in the user's browser.

The HTML file after modification is listed in Listing 22.2.

LISTING 22.2 The Modified HTML Page

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Current Date and Time</title>
    <style>
        p {font: 14px normal arial, verdana, helvetica;}
    </style>
    <script src="datetime.js"></script>
</head>
<body>
    The current date and time are:<br/>
    <div id="div1"></div>
    <input id="btn1" type="button" value="Refresh" />
    <noscript>
        <p>Your browser does not support JavaScript.</p>
        <p>Please consult your computer's operating system for local date
and time

information or click <a href="clock.php" target="_blank">HERE</a> to read
the server
```

```
time.</p>
    </noscript>
</body>
</html>
```

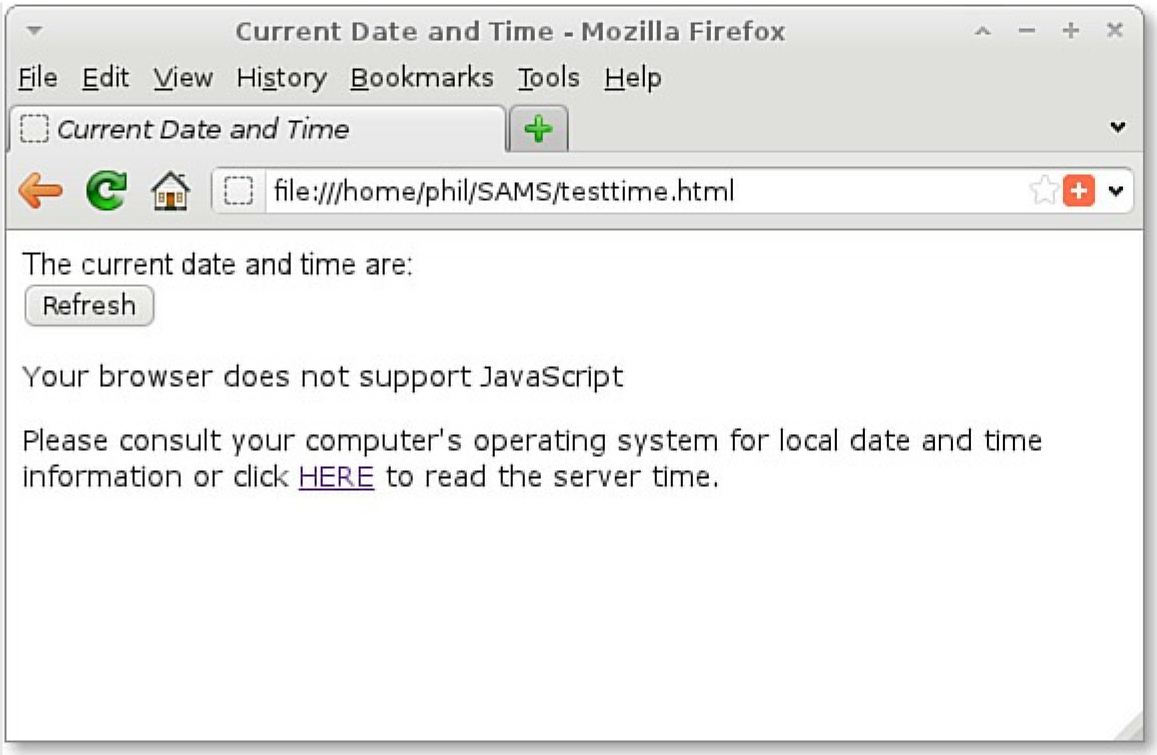
Within our JavaScript source file telltime.js, we use `window.onload` to add the event listener for the button. Finally we call `telltime()` to generate the date and time information to display on the page. The JavaScript code is shown in Listing 22.3.

LISTING 22.3 datetime.js

Click here to view code image

```
function telltime() {
    var out = "";
    var now = new Date();
    out += "<br />Date: " + now.getDate();
    out += "<br />Month: " + now.getMonth();
    out += "<br />Year: " + now.getFullYear();
    out += "<br />Hours: " + now.getHours();
    out += "<br />Minutes: " + now.getMinutes();
    out += "<br />Seconds: " + now.getSeconds();
    document.getElementById("div1").innerHTML = out;
}

window.onload = function() {
    document.getElementById("btn1").onclick= function()
{location.reload();}
    telltime();
}
```

With JavaScript enabled, the script works just as it did in Hour 4. However, with JavaScript disabled, the user now sees the page as shown in Figure 22.1.


Current Date and Time - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Current Date and Time
file:///home/phil/SAMS/testtime.html
Refresh

The current date and time are:

Your browser does not support JavaScript

Please consult your computer's operating system for local date and time information or click HERE to read the server time.

FIGURE 22.1 Extra information for users without JavaScript

Summary

In this hour we rounded up and presented various examples of good practice in writing JavaScript. Used together they should help you deliver your code projects more quickly, with higher quality and much easier maintenance.

Q&A

Q. Why would a user turn off JavaScript?

A. Remember that the browser might have been set up by the service provider or employer with JavaScript turned off by default, in an effort to improve security. This is particularly likely in an environment such as a school or an Internet cafe.

Additionally, some corporate firewalls, ad-blocking, and personal antivirus software prevent JavaScript from running, and some mobile devices have web browsers without complete JavaScript support.

Q. Are there any other options besides `<noscript>` for dealing with users who don't have JavaScript enabled?

A. An alternative that avoids `<noscript>` is to send users who *do* have JavaScript support to an alternative page containing JavaScript-powered enhancements:

Click here to view code image

```
<script>window.location="enhancedPage.html";</script>
```

If JavaScript is available and activated, the script redirects the user to the enhanced page. If the browser doesn't have JavaScript support, the script won't be executed, and the user is left viewing the more basic version.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. The modularization of code into reusable blocks for more general use is called:
   a. Abstraction
   b. Inheritance
   c. Unobtrusive JavaScript
2. The CSS for your page should be confined as much as possible to the:
   a. Semantic layer
   b. Presentation layer
   c. Behavior layer
3. Unobtrusive JavaScript code should, wherever possible, be placed
   a. In an external file
   b. Between `<script>` and `</script>` tags in the page `<head>`
   c. Inline

Answers

1. a. Abstraction
2. b. Where possible, all CSS goes in the presentation layer.
3. a. Use external JavaScript files where it's feasible to do so.

Exercises

- Pick some Try It Yourself sections from earlier in the book and see what you can do to make the code more unobtrusive, without adversely affecting the script's operation.
- Can you work out how to further modify the code of Listing 22.2 and Listing 22.3 to ensure that users without JavaScript enabled see just the content of the `<noscript>` tag, without the additional text and button being present? (Hint: Write these items to the page with innerHTML or via DOM methods.)

Hour 23. Debugging Your Code

What You'll Learn in This Hour:

- The types of errors common in JavaScript code
- How to carry out simple debugging with `alert()`
- Using the browser console and `console.log()`
- Grouping messages in the console
- Using breakpoints

As you delve into more advanced scripting, you're going to now and then create JavaScript programs that contain errors.

JavaScript errors can be caused by a variety of minor blunders, such as mismatched opening and closing parentheses, mistyping of variable names or keywords, making calls to nonexistent methods, and so on. This hour aims to offer some straightforward tips and suggestions for diagnosing errors and correcting your code, making your programming hours more pleasurable and productive.

An Introduction to Debugging

The process of locating and correcting bugs is known as *debugging*, and it can be one of the most tricky and frustrating parts of the development process.

Types of Errors

The errors that can crop up in your code usually conform to one of three types:

- **Syntax errors**—These can include typographical and spelling errors, missing or mismatched quote marks, missing or mismatched parentheses/braces, and case-sensitivity errors.
- **Runtime errors**—Errors that occur when the JavaScript interpreter tries to do something it can't make sense of. Examples include trying to treat a string as if it were a numerical value and trying to divide a number by zero.
- **Faulty program logic**—These mistakes don't always generate error messages—the code may be perfectly valid—but your script doesn't do what you want it to. These are usually problems associated with algorithms or logical flow within the script.

Choosing a Programmer's Editor

Whatever platform you work on, and whatever your browser of choice, it makes sense to have a good code editor. While it's certainly possible to write code in simple programs like the Windows Notepad text editor, a dedicated editor makes life a lot easier.

Many such programs are available, often free of charge under open source and similar licenses. Here I list a small selection of no-cost editors, but look around for one that suits your platform, your working style, and your pocket.

- Notepad++ (Windows)
- JEdit (should work on any platform that has Java installed)
- PSPad (Windows)
- JuffEd (Windows, Linux)
- Geany (Windows, Linux)

Editors offer a range of features and capabilities, but as a minimum I would suggest looking for an editor with the following:

- **Line numbering**—This is especially useful if you store your JavaScript in external files (and is yet another reason you should do so, wherever feasible). That way the line numbers of any error messages generated by your browser's debugger will usually match those in the source file open in the editor.
- **Syntax highlighting**—When you become familiar with your editor's scheme of syntax highlighting, you can on many occasions spot coding errors simply because the code in the editor "looks wrong." It's surprising how quickly you get used to the colors of keywords, variables, string literals, objects, and so on in your favorite editing program. Many editors let you alter the syntax highlighting color scheme to your own taste.
- **Parentheses matching**—As an error-seeking missile, parentheses matching is invaluable. Good editors will show matching pairs of open/close occurrences and for all types of brackets, braces, and parentheses. When your code has several levels of nested parentheses it's easy to lose count.
- **Code completion or tooltip-style syntax help**—Some editors offer pop-up tooltip-style help for command functions and expressions. This can save you having to take your eyes from the editor window to look up an external reference.

Simple Debugging with `alert()`

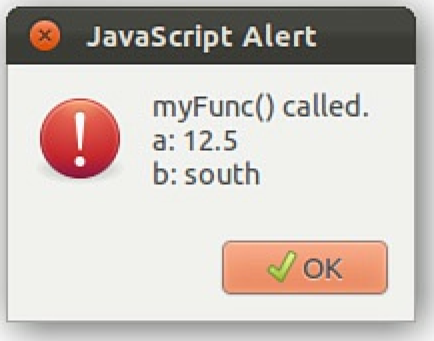
Sometimes you want a really simple and quick way to read a variable's value, or to track the order in which your code executes.

Perhaps the easiest way of all is to insert JavaScript `alert()` statements at appropriate points in the code. Let's suppose you want to know whether an apparently unresponsive function is actually being called, and if so, with what parameters:

Click here to view code image

```
    function myFunc(a, b) {
       alert("myFunc() called.\na: " + a + "\nb: " + b);
       // .. rest of function code here ...
    ...}
```

When the function is called at runtime, the `alert()` method executes, producing a dialog like the one in Figure 23.1.



myFunc() called.
a: 12.5
b: south

FIGURE 23.1 Using a JavaScript `alert()`

Remember to put a little more information in the displayed message than just a variable value or one-word comment; in the heat of battle, you'll likely forget to what variable or property the value in the `alert()` refers.

More Advanced Debugging

Placing `alert()` calls in your code is perhaps OK for a quick-and-dirty debug of a short piece of code. The technique does, however, have some serious drawbacks:

- You have to click OK on each dialog to allow processing to continue. This can be demoralizing, especially when dealing with long loops!
- The messages received are not stored anywhere, and disappear when the dialog is cleared; you can't go back later and review what was reported.
- You need to go back into the editor and erase all the `alert()` calls before your code can "go live."

In this section, we'll look at some more advanced debugging techniques.

The Console

Thankfully, most modern browsers provide a JavaScript Console that you can use to better effect for logging debugging messages. How to open the console varies from browser to browser:

- In Internet Explorer, to open the Developer Tools: F12

- For Chrome’s Developer Tools and Opera’s Dragonfly Debugger: Ctrl+Shift+I
- Using Firefox with the Firebug extension: F12

The examples in this section assume that you’re using one of the previous debuggers. If not, you may have to consult your debugger’s documentation to see how to carry out some of the tasks I describe. How your browser presents such errors to you differs from browser to browser.

Try it Yourself: Using Your Browser’s Debugging Tools

Have a look at the code in Listing 23.1.

LISTING 23.1 A Program with Errors

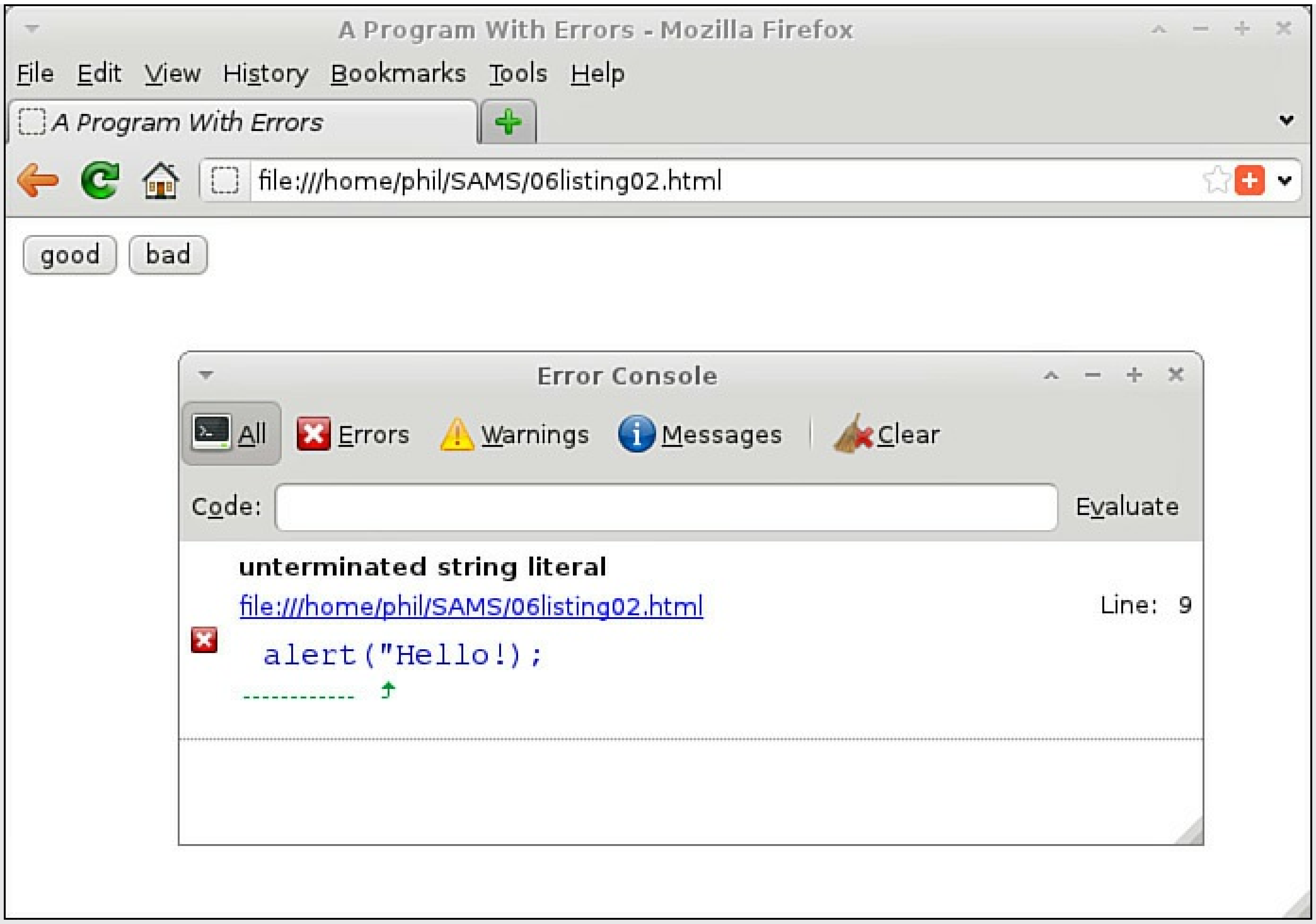
Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Strings and Arrays</title>
</head>
<body>
    <script>
        function sayHi() {
            alert("Hello!);
        }
    </script>
    <input type="button" value="good" onclick="sayHi()" />
    <input type="button" value="bad" onclick="sayhi()" />
</body>
</html>
```

This code listing has two different types of errors. First, in the call to the `alert()` method, our argument is missing its closing quotation mark.

Second, the `onclick` handler of the second button calls the function `sayhi()`—remember that function names are case sensitive, so in fact there is no function defined with the name `sayhi()`.

Loading the page into Firefox, we can see the expected two buttons, one labeled “good” and the other “bad.” Neither seems to do anything. I can open Firefox’s Error Console by pressing Ctrl+Shft+J, and the result is shown in Figure 23.2.


A Program With Errors - Mozilla Firefox
File Edit View History Bookmarks Tools Help
A Program With Errors
file:///home/phil/SAMS/06listing02.html
good
bad
Error Console
All
Errors
Warnings
Messages
Clear
Code:
Evaluate

unterminated string literal
file:///home/phil/SAMS/06listing02.html Line: 9

```
alert("Hello!);
```

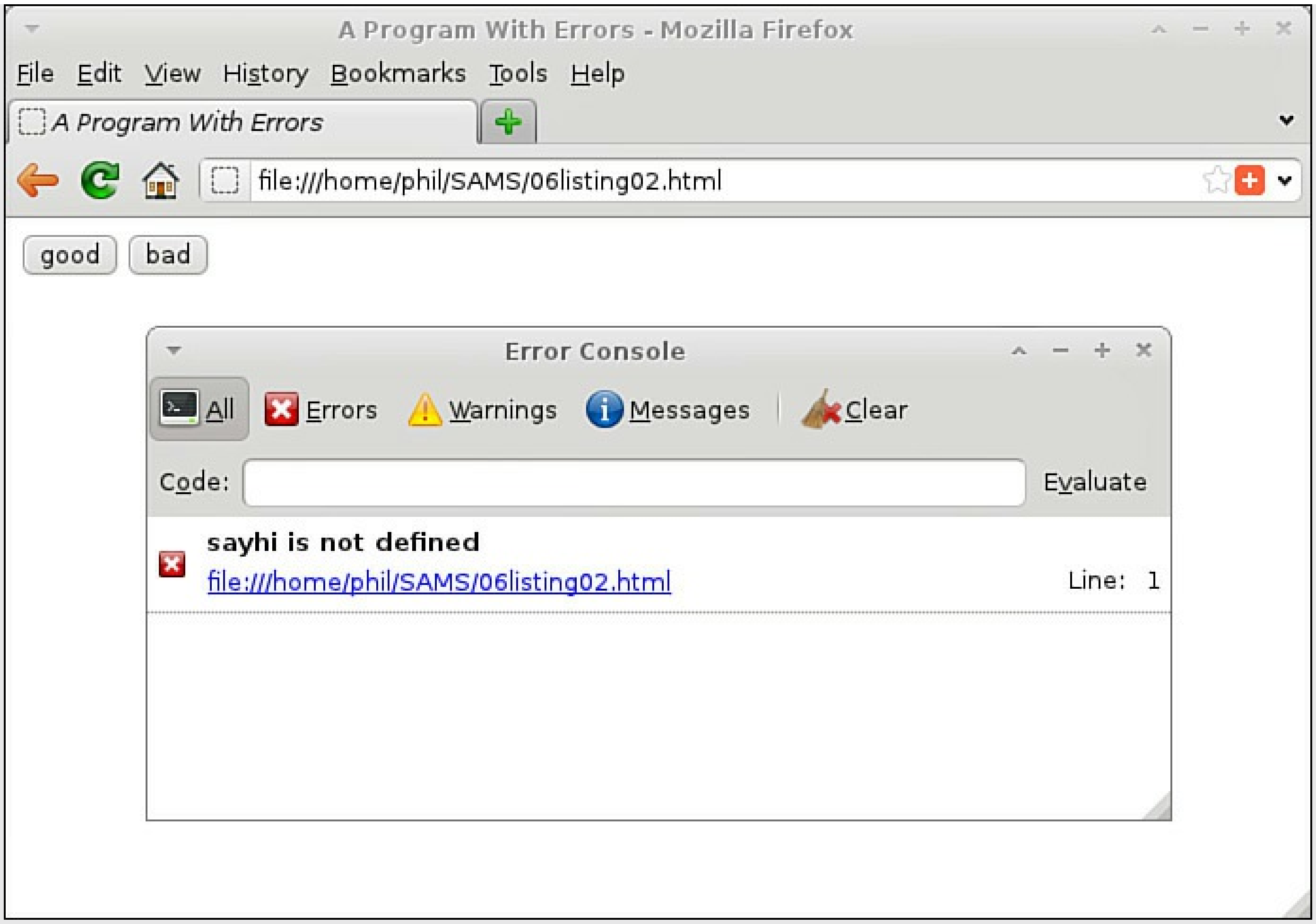
FIGURE 23.2 The Firefox Error Console

That's a helpful start. Firefox tells me it found an *unterminated string literal*, gives me the line number, and even shows me the line of code with an arrow pointed at the section where it has a problem.

With the error corrected and the file saved again, I'm ready to try again. First I click Clear on the toolbar of the Error Console to remove the old error message; then I reload my test page.

That looks better. My page comes up again, and the Error Console stays blank. Clicking on the button labeled "good" opens the expected `alert()` dialog—so far, so good.

But clicking on the button labeled "bad" doesn't seem to do anything—so I refer again to the Error Console, as shown in Figure 23.3.

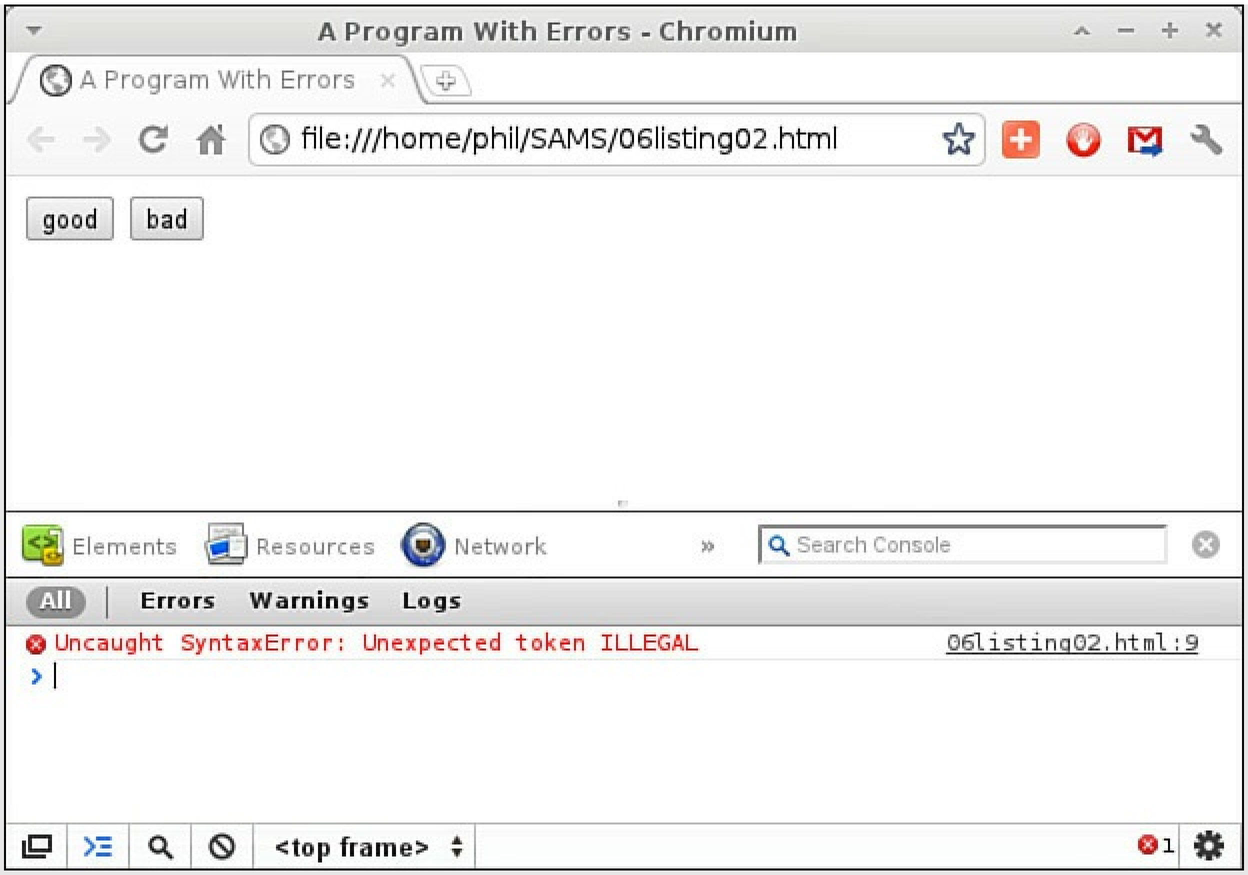

A Program With Errors - Mozilla Firefox
File Edit View History Bookmarks Tools Help
A Program With Errors
file:///home/phil/SAMS/06listing02.html
good
bad
Error Console
All
Errors
Warnings
Messages
Clear
Code:
Evaluate

```
sayhi is not defined
file:///home/phil/SAMS/06listing02.html                Line: 1
```

FIGURE 23.3 The second error

Firefox again identifies the problem: "sayhi is not defined." Now we're well on the way to having our code fully debugged and working correctly.

Every browser has its own way of dealing with errors. Figure 23.4 shows how the Chromium browser reports the initial error of the unterminated string literal.



```
Uncaught SyntaxError: Unexpected token ILLEGAL          06listing02.html:9
```

FIGURE 23.4 Google Chromium JavaScript console

Note

Google Chrome and Chromium are almost identical browsers, differing mainly in how they are packaged and distributed. Essentially, Google Chrome is the Chromium open source browser packaged and distributed by Google.

Chromium's message is a slightly more cryptic "Uncaught syntax error: Unexpected token ILLEGAL," but it also gives the line number in a clickable link that shows me the faulty line of code.

To open Internet Explorer's developer tools, press F12 or select Developer Tools from the IE9 Tools menu. Select the Console tab to view error messages returned by JavaScript.

Tip

It's worth getting to know the debugging tools in your favorite browser, and even think about switching browsers if you particularly prefer the tools on offer in another. If you plan to regularly write JavaScript code, it makes sense to do so in a development environment in which you feel comfortable, where you'll be more productive and less frustrated.

Try it Yourself: A Banner-Cycling Script

Let's put to use some of what you learned in this hour by writing a script to cycle images on the page. I'm sure you have seen this sort of program before, either as an image slideshow, or perhaps to rotate advertisement banners.

First, I want to introduce you to two new items. The first is an event handler that you haven't met before—the `onLoad` method of the `window` object. Its operation is simple: We can attach it to the `<body>` element like this:

Click here to view code image

```
<body onload="somefunction()" >
```

When the page has finished loading completely, the `onLoad` event fires, and the code specified in the event handler runs. We use this event handler to run our banner rotator as soon as the page has loaded.

Second, we are going to use JavaScript's `setInterval()` function. This function allows us to run a JavaScript function repeatedly, with a preset delay between successive executions.

The `setInterval()` function takes two arguments. The first is the name of the function we want to run, the second the delay (in milliseconds) between successive executions of the function. As an example, the line

```
setInterval(myFunc, 5000);
```

would execute the function `myFunc()` every five seconds.

We use `setInterval()` to rotate the banner image at a regular interval.

Create a new file named banner.html and enter the code from Listing 23.2.

LISTING 23.2 A Banner Rotator

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
    <title>Banner Cycler</title>
    <script>
        var banners = ["banner1.jpg", "banner2.jpg", "banner3.jpg"];
        var counter = 0;
        function run() {
            setInterval(cycle, 2000);
        }
        function cycle() {
            counter++;
            if(counter == banners.length) counter = 0;
            document.getElementById("banner").src = banners[counter];
        }
    </script>
</head>
<body onload = "run();">
    <img id="banner" alt="banner" src="banner1.jpg" />
</body>
</html>
```

The HTML part of the page could hardly be simpler—the body of the page just contains an image element. This image will form the banner, which will be "rotated" by changing its `src` property.

Now let's take a look at the code.

The function `run()` contains only one statement, the `setInterval()` function. This function executes another function, `cycle()`, every two seconds (2000 milliseconds).

Every time the function `cycle()` executes, we carry out three operations:

1. Increment a counter.

   ```
   counter++;
   ```

2. Use a conditional statement to check whether the counter has reached the number of elements in the array of image names; if so, reset the counter to zero.

   Click here to view code image

   ```
   if(counter == banners.length) counter = 0;
   ```

3. Set the `src` property of the displayed image to the appropriate file name selected from the array of images file names.

   Click here to view code image

   ```
   document.getElementById("banner").src = banners[counter];
   ```

The operation of the script is shown in Figure 23.5.

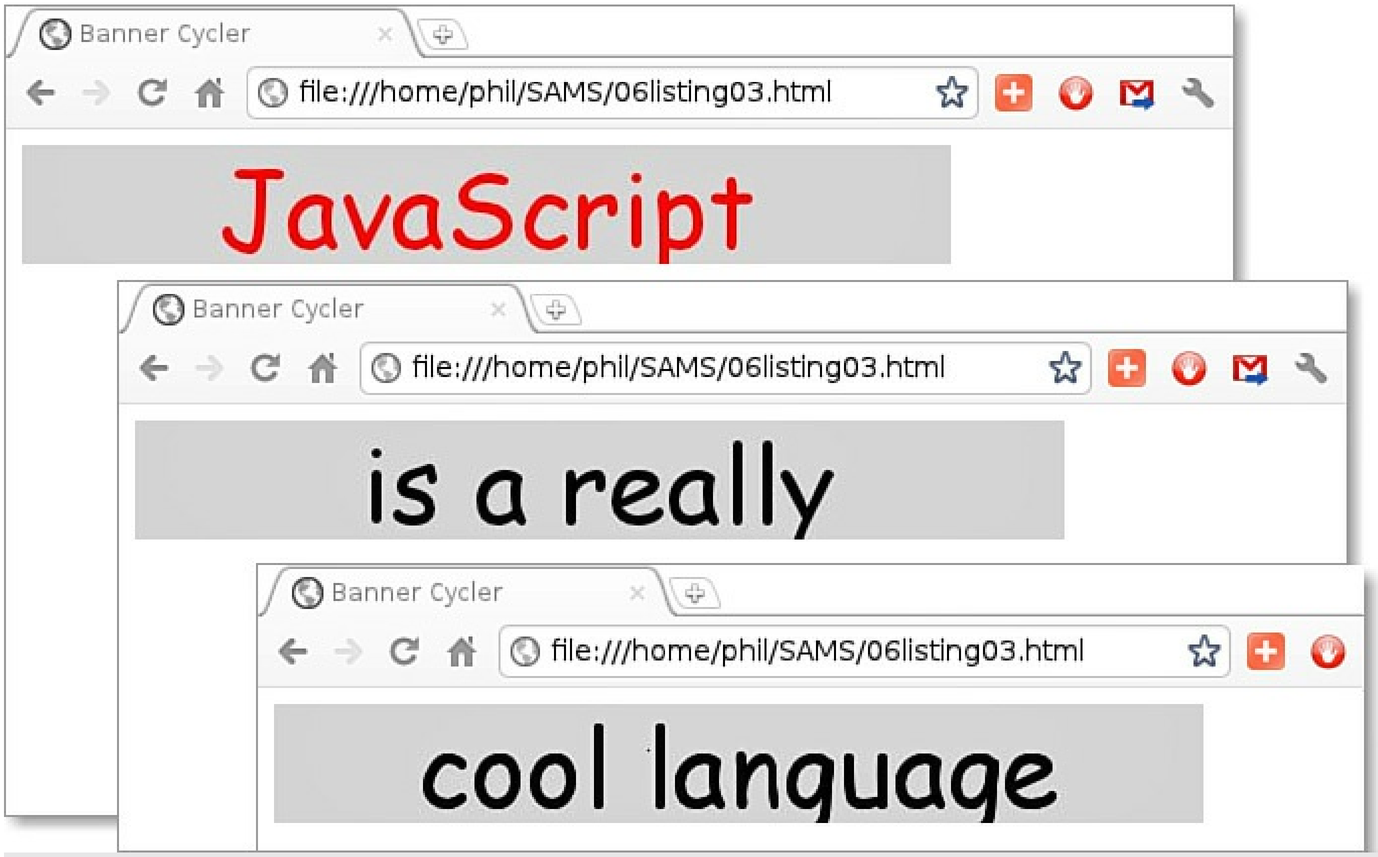
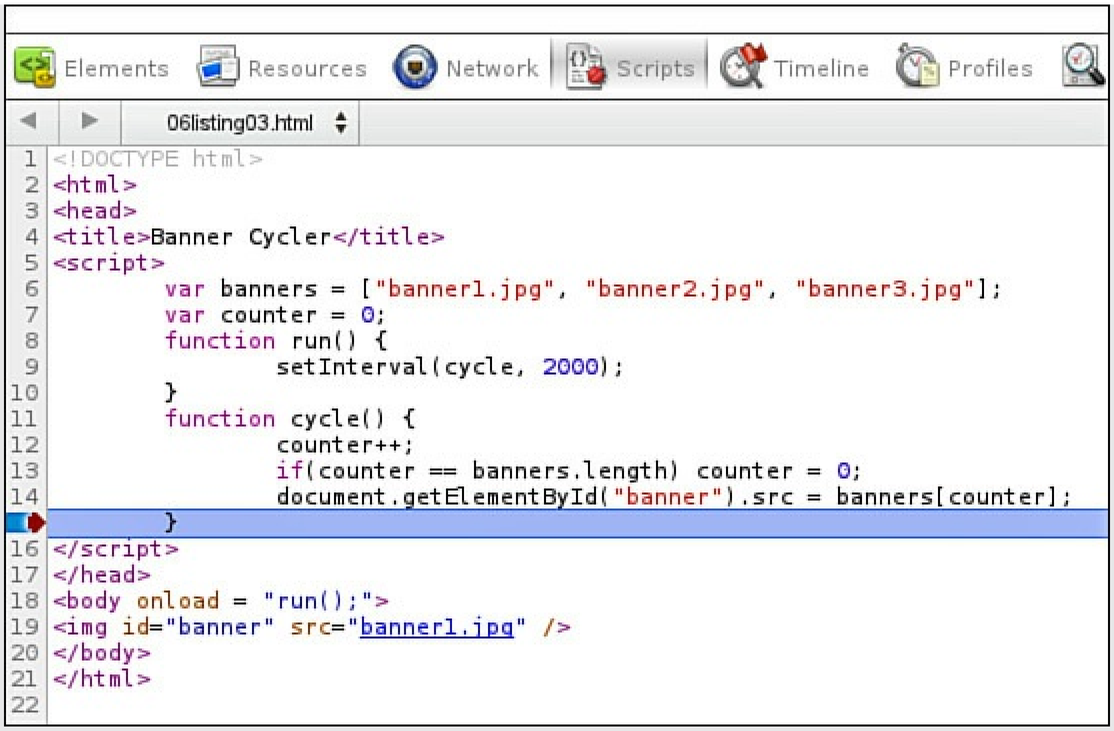

Banner Cycler
file:///home/phil/SAMS/06listing03.html
JavaScript
Banner Cycler
file:///home/phil/SAMS/06listing03.html
is a really
Banner Cycler
file:///home/phil/SAMS/06listing03.html
cool language

FIGURE 23.5 Our banner cycler

Now let's examine the script operation using browser-based debug tools. I'm using Chromium, so I open the Developer Tools console again like I did in Figure 23.4. In Chromium that's **Settings > More Tools > Developer Tools** or the shortcut Ctrl+Shift+I.

This time I select the Scripts tab in the lower pane. To the left of the lower pane, the code is listed; I'm going to click on the line number of line 15 to set a breakpoint, as shown in Figure 23.6.



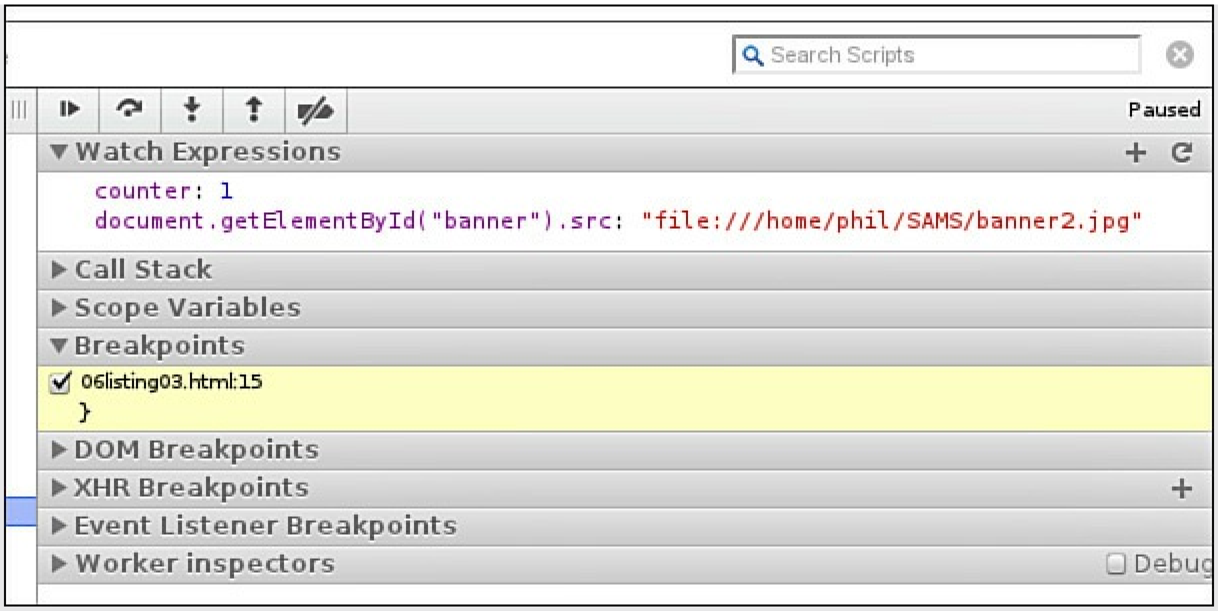
```
<!DOCTYPE html>
<html>
<head>
<title>Banner Cycler</title>
<script>
        var banners = ["banner1.jpg", "banner2.jpg", "banner3.jpg"];
        var counter = 0;
        function run() {
                setInterval(cycle, 2000);
        }
        function cycle() {
                counter++;
                if(counter == banners.length) counter = 0;
                document.getElementById("banner").src = banners[counter];
        }
</script>
</head>
<body onload = "run();">
<img id="banner" src="banner1.jpg" />
</body>
</html>
```

FIGURE 23.6 Setting a breakpoint

While this breakpoint remains set, code execution will halt every time this line of code is reached, before executing the code in the line—in this case, before completing the current execution of the function `cycle()`.

On the right-hand side of the same pane, our breakpoint now appears in the Breakpoints panel. In the same pane, I can click in the Watch Expressions panel and add the names of any variables or expressions whose values I want to examine each time the program pauses; I'm going to enter `counter` and `getElementById("Banner").src` to see what values they contain.

Figure 23.7 shows the display when the program next pauses, showing the values of my two chosen expressions.



```
counter: 1
document.getElementById("banner").src: "file:///home/phil/SAMS/banner2.jpg"
```

```
06listing03.html:15
}
```

FIGURE 23.7 Showing variable values at a breakpoint

Pressing the Play icon above the panel allows the script to restart.

Try using your own browser's debugging tools to explore the program's operation.

Tip

I have only scratched the surface here of the capabilities of the debugger in Google Chrome/Chromium. To learn more, there is a good tutorial at https://developer.chrome.com/devtools/docs/javascript-debugging to get you started.

If Firefox is your browser of choice for development work, you would do well to install the popular Firebug extension, which you can read about at http://getfirebug.com/javascript and which has broadly similar capabilities.

Those using Microsoft Internet Explorer will find good information on debugging with the F12 Developer Tools at https://msdn.microsoft.com/en-us/library/ie/gg589512(v=vs.85).aspx.

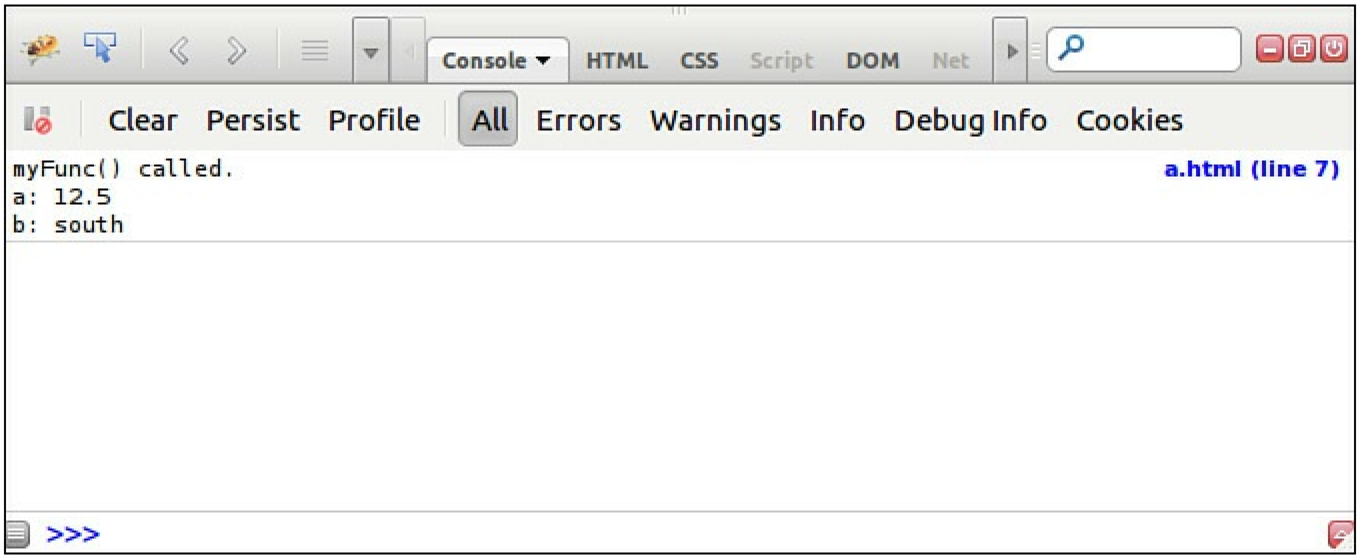
Opera contains the Dragonfly debugging tool, which you can read about at http://www.opera.com/dragonfly/documentation/.

The console provides a number of methods you can use in your code in place of the cumbersome and limited `alert()` call, perhaps the most well-known being `console.log()`:

Click here to view code image

```
    function myFunc(a, b) {
        console.log("myFunc() called.\na: " + a + "\nb: " + b);
        // .. rest of function code here ...
    ...}
```

Rather than interrupt program operation, `console.log()` operates invisibly to the user unless he or she happens to be looking at the console. Figure 23.8 shows the result of running the preceding code with the console open in Firefox with the Firebug extension installed.

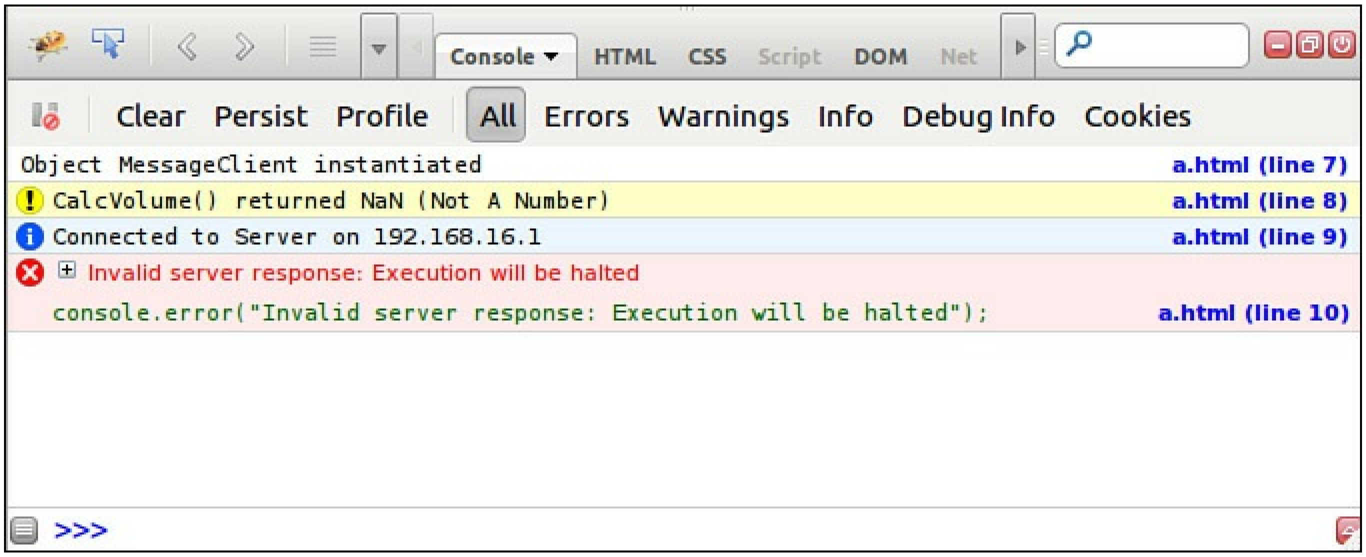


```
myFunc() called.                                    a.html (line 7)
a: 12.5
b: south
```

FIGURE 23.8 A message logged in the console

In addition to `console.log()`, you can also take advantage of `console.warn()`, `console.info()`, and `console.error()`. These all record messages at the console in slightly different styles, allowing you to build up a picture of how your script is running.

Figure 23.9 shows how Firebug's console displays each one; the display will be slightly different in other browsers.



```
Object MessageClient instantiated                                    a.html (line 7)
CalcVolume() returned NaN (Not A Number)                             a.html (line 8)
Connected to Server on 192.168.16.1                                  a.html (line 9)
Invalid server response: Execution will be halted
console.error("Invalid server response: Execution will be halted");  a.html (line 10)
```

FIGURE 23.9 Different types of console messages

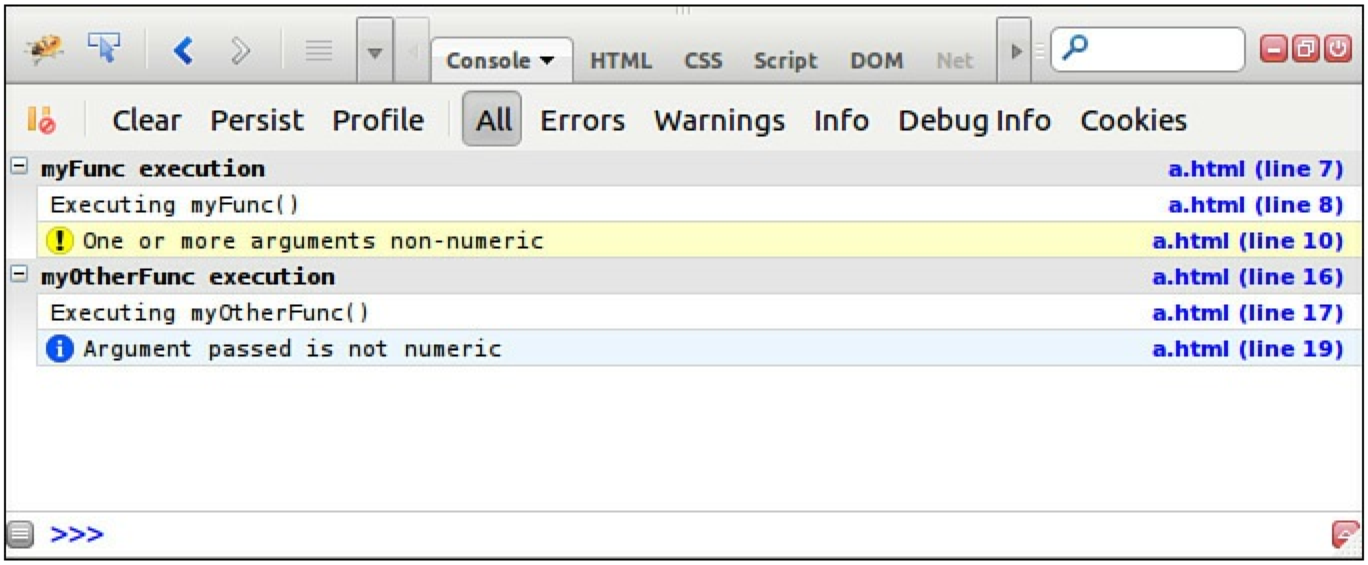
Grouping Messages

Sorting console debugging messages into groups makes them even more readable. You can name the individual message groups any way you like:

Click here to view code image

```
function myFunc(a, b) {
   console.group("myFunc execution");
   console.log("Executing myFunc()");
   if(isNaN(a) || isNaN(b)) {
      console.warn("One or more arguments non-numeric");
   }
   console.groupEnd();
   myOtherFunc(a+b);
}

function myOtherFunc(c) {
   console.group("myOtherFunc execution");
   console.log("Executing myOtherFunc()");
   if(isNaN(c)) {
      console.info("Argument is not numeric");
   }
   console.groupEnd();
   // .. rest of function code here ...
}
```

In this code snippet I've defined two `console.group()` sections, and named them to associate them with the functions in which they execute. Each group ends with a `console.groupEnd()` statement. When the code runs, any console messages display in groups, as shown in Figure 23.10.



```
myFunc execution                                a.html (line 7)
  Executing myFunc()                            a.html (line 8)
  One or more arguments non-numeric             a.html (line 10)
myOtherFunc execution                           a.html (line 16)
  Executing myOtherFunc()                       a.html (line 17)
  Argument passed is not numeric                a.html (line 19)
```

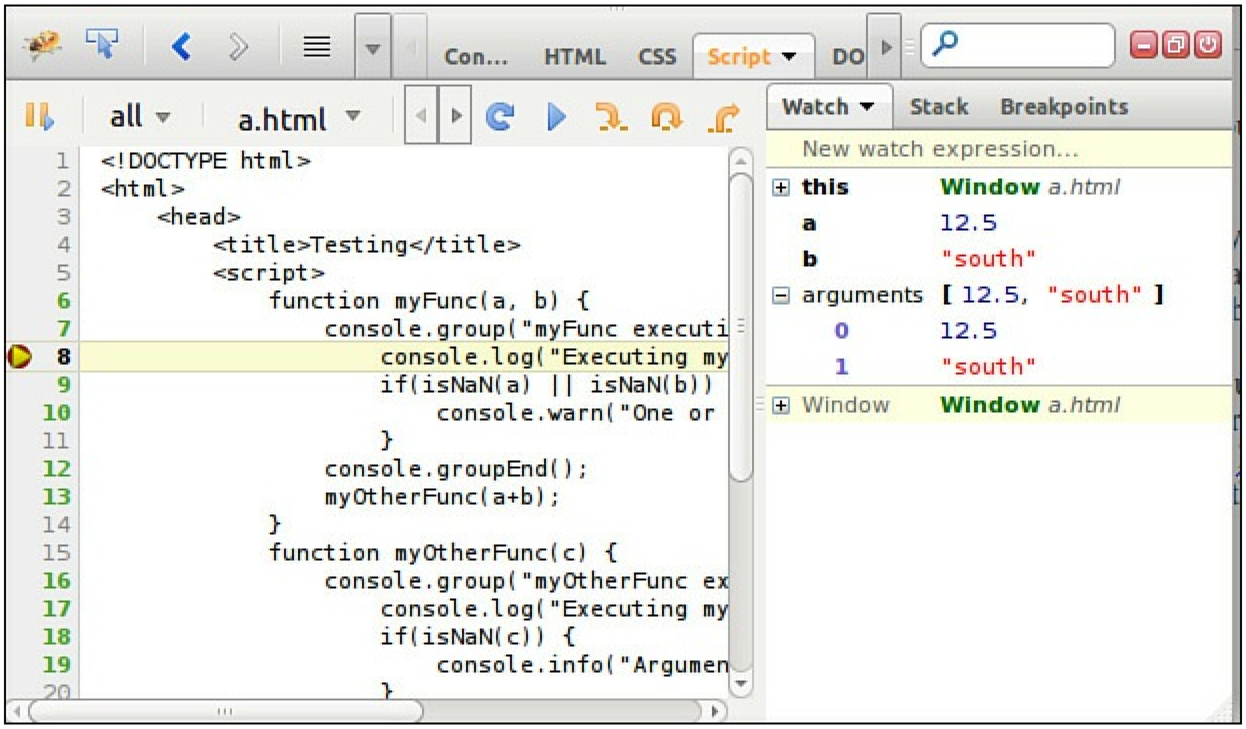
FIGURE 23.10 Grouped messages

Using Breakpoints to Halt Code Execution

As your scripts grow in complexity, you're likely to find that even console logging isn't enough to let you debug effectively.

To perform more detailed debugging, you can set so-called breakpoints in the code at places of interest. When code execution arrives at a breakpoint, it pauses; while time remains frozen you can examine how your code is operating, check variable values, read logged messages, and so on.

To set a breakpoint in most popular debuggers you need to go to the Scripts panel, where you'll see your code listed. Click on a line number (or just to the left of it) to set a breakpoint at that line. In Figure 23.11, a breakpoint has been set on line 8 of the code. The execution has stopped at this point, and you can see the current values of the individual variables in the right panel. You can remove breakpoints by clicking again on the breakpoint icon in the left margin.



```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Testing</title>
5          <script>
6              function myFunc(a, b) {
7                  console.group("myFunc executi
8                      console.log("Executing my
9                      if(isNaN(a) || isNaN(b))
10                         console.warn("One or
11                     }
12                 console.groupEnd();
13                 myOtherFunc(a+b);
14             }
15             function myOtherFunc(c) {
16                 console.group("myOtherFunc ex
17                     console.log("Executing my
18                     if(isNaN(c)) {
19                         console.info("Argumen
20                     }
```

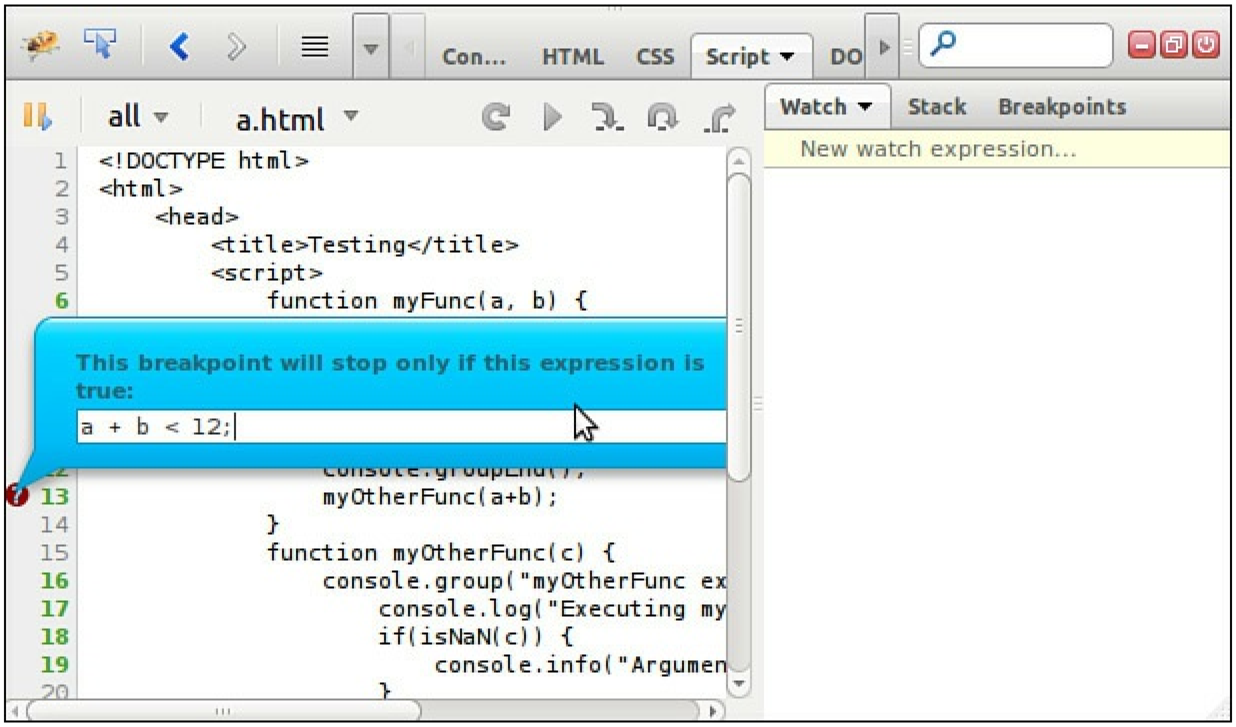
```
⊞ this        Window a.html
    a         12.5
    b         "south"
⊟ arguments   [ 12.5, "south" ]
     0        12.5
     1        "south"
⊞ Window      Window a.html
```

FIGURE 23.11 Execution stopped at a breakpoint

Conditional Breakpoints

Sometimes it helps to break code execution only when a particular situation occurs. You can set a conditional breakpoint by right-clicking the breakpoint icon in the left column and entering a conditional statement.

Your code will execute without interruption until the condition is fulfilled, at which point execution will halt. For instance, in Figure 23.12, the code will halt if the sum of a and b is less than 12. You can edit the expression at any time just by right-clicking once more on the breakpoint icon.



```
<!DOCTYPE html>
<html>
    <head>
        <title>Testing</title>
        <script>
            function myFunc(a, b) {
```

This breakpoint will stop only if this expression is true:

```
a + b < 12;
```

```
                console.groupEnd();
                myOtherFunc(a+b);
            }
            function myOtherFunc(c) {
                console.group("myOtherFunc ex
                    console.log("Executing my
                    if(isNaN(c)) {
                        console.info("Argumen
                    }
```

FIGURE 23.12 A conditional breakpoint

When code execution halts at a breakpoint you can choose to continue code execution, or step through your code one statement at a time, by using one of the code execution buttons; these usually look something like VCR controls, and appear at the top of one of the panels in the debugger. In most debuggers, the options are:

- **Continue**—Resume execution and only pause again if/when another breakpoint is reached.
- **Step Over**—Execute the current line, including any functions that are called, then move to the next line.
- **Step Into**—Move to the next line, as with Step Over, unless the line calls a function; in that case jump to the first line of the function.
- **Step Out**—Leave the current function and return to the place from which it was called.

Launching the Debugger from Your Code

It's also possible, and often useful, to set breakpoints from within the JavaScript code. We can do this by using the keyword `debugger`:

Click here to view code image

```
function myFunc(a, b) {
   if(isNaN(a) || isNaN(b)) {
   debugger;
   }
// .. rest of function code here ...
}
```

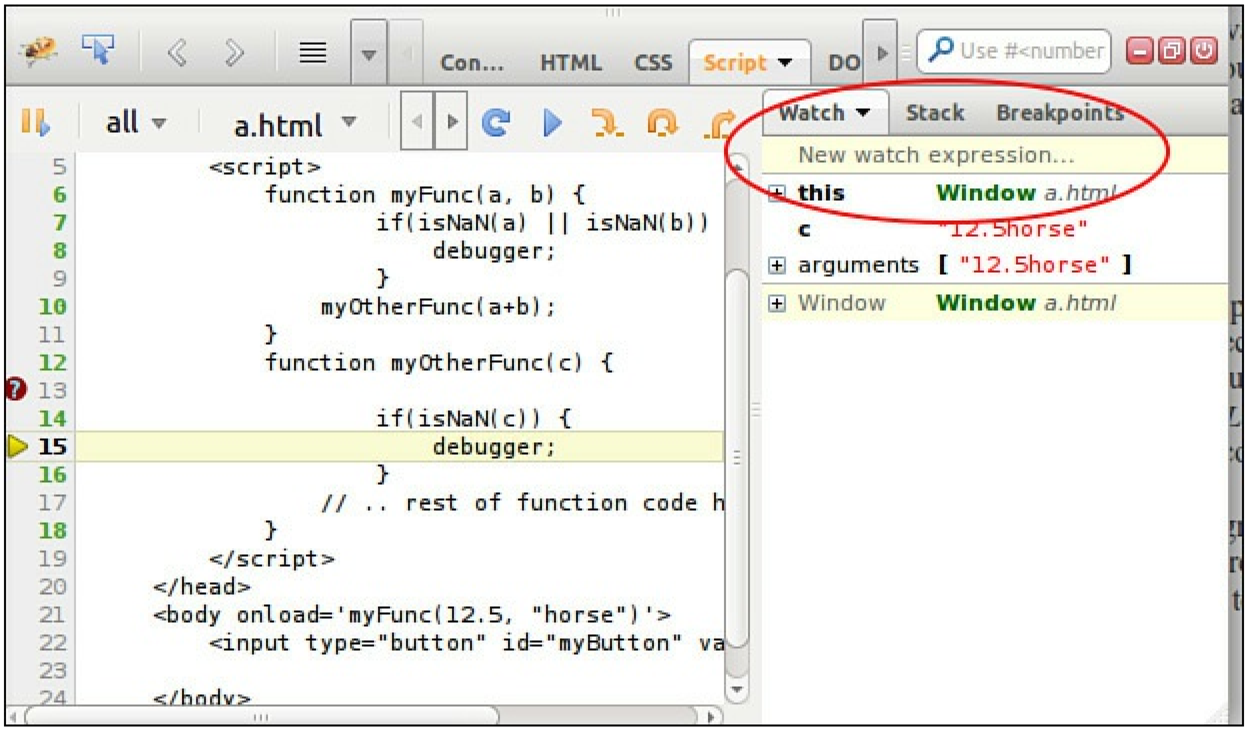
In this example, code execution will be halted and the debugger opened only if the conditional expression evaluates to true.

The debugging tools allow you to halt code execution in other circumstances too, such as when the DOM has been altered, or when an uncaught exception has been detected, but these are more advanced cases outside the scope of this discussion.

Watch Expressions

A watch expression is a valid JavaScript expression that the debugger continuously evaluates, making the value available for you to inspect. Any valid expression can be used, ranging from a simple variable name to a formula containing logical and arithmetic expressions or calls to other functions.

You can enter a new watch expression via the right-hand panel of the Script tab, as shown in Figure 23.13 (Firefox/Firebug).



```
        <script>
            function myFunc(a, b) {
                    if(isNaN(a) || isNaN(b))
                        debugger;
                    }
                myOtherFunc(a+b);
            }
            function myOtherFunc(c) {

                    if(isNaN(c)) {
                        debugger;
                    }
                // .. rest of function code h
            }
        </script>
    </head>
    <body onload='myFunc(12.5, "horse")'>
        <input type="button" id="myButton" va

    </body>
```

```
New watch expression...
this        Window a.html
c           "12.5horse"
arguments   [ "12.5horse" ]
Window      Window a.html
```

FIGURE 23.13 A watch expression

Validating JavaScript

A different and complementary approach to checking your JavaScript code for problems is to use a validation program. This will check that it conforms to the correct syntax rules of the language. These programs are sometimes bundled with commercial JavaScript editors, or you can simply use Douglas Crockford’s JavaScript Lint, which is available free online at http://www.jslint.com/.

Here you can simply paste your code into the displayed window and click the button.

Don’t be too dismayed if the program reports a lot of errors—just work through them one at a time. JSLint is very thorough and will even report various issues of coding style that wouldn't affect your code's running at all, but do help to improve how you program!

Summary

In this hour you learned a lot about debugging your JavaScript code, including using the browser console, as well as setting breakpoints and stepping through code in the debugger.

Q&A

Q. How should I choose a programmer's editor?

A. It's completely up to personal choice. Many are free or have a free version, so there's nothing to stop you from trying several before deciding.

Q. Where can I find out more about JavaScript debugging?

A. Many tutorials exist online. Start with the one published by W3Schools, at http://www.w3schools.com/js/js_debugging.asp.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. Which of these error types are you not likely to find in your JavaScript programs?
   a. Syntax errors
   b. Compilation errors
   c. Runtime errors
2. What does a breakpoint do?
   a. Pauses code execution at a given place
   b. Causes JavaScript to step out of a loop
   c. Produces a JavaScript error
3. What line in your code will launch the debugger?
   a. `debug;`
   b. `debugger;`
   c. `pause;`

Answers

1. b. JavaScript is an interpreted, rather than a compiled, language, so you won’t encounter compilation errors.
2. a. Pauses code execution at a given place.
3. b. To launch the debugger, type

```
debugger;
```

Exercises

- How would you modify the banner-cycling script to add links to the banners, such that each image displayed linked to a different external page?
- Using your knowledge of random number generation using the Math object, can you rewrite the banner-cycling script to show a random banner at each change, instead of cycling through them in order? Use your browser’s built-in debugging tools to help you.

Hour 24. JavaScript Unit Testing

What You'll Learn in This Hour:

- What unit testing is and why it's used
- How to make your JavaScript code more easily testable
- Some examples of JavaScript test suites
- About the QUnit test suite and the CommonJS unit testing specification

Often, a little JavaScript hack that started out as just a few lines of code subsequently grows to ten, then twenty, then fifty. Meanwhile, functions are tweaked to do a little more, conditional statements gain a few extra conditions, or a couple extra variables are created.

When such an application (inevitably) breaks, it can be a nightmare to unravel the code and find the problem.

As you already read in Hour 22, "Good Coding Practice," good coding practice can help to make your code easier to understand and to maintain, but there's something else you can do, too. As your JavaScript applications grow in size, complexity, and sophistication, it becomes even more important to write code that can be easily tested.

In this hour you'll learn some ways to write your code to make it suitable for unit testing, and how to write and perform such tests.

What Is Unit Testing?

If you're unfamiliar with the concept of unit testing, don't worry, as it's not too complicated to understand.

Usually when somebody thinks about testing a JavaScript application, they imagine testing the completed system to see if it works as expected. This is essentially a test to see if the various pieces of the application work correctly together, and is often known as *integration testing*.

Unit testing is a software verification and validation method in which an application is broken down into its smallest testable parts, called *units*, then these units are individually examined and tested for proper operation. A unit is the smallest testable part of an application, such as an individual function or method.

Unit testing can be done manually, but such testing is usually automated.

Essentially, you write a series of tests for each fundamental element of your code, to test that unit's performance under all conceivable types of input. If all of these tests are passed, you can be confident that each tested element is fit for purpose.

Try it Yourself: A Home-Cooked Unit Test

Let's jump right in and perform a unit test from scratch.

In this example we'll write some tests, perform them on a sample unit, and send a summary of the results to the browser console.

Let's look at a function we wrote way back in Hour 3, "Using Functions," to add tax at a given percentage to a net figure and return the gross amount:

Click here to view code image

```
function addTax(subtotal, taxRate) {
    var total = subtotal * (1 + (taxRate/100));
    return total;
}
```

This function forms the unit that we'll subject to some tests.

We'll save this function in a file tax.js, and include it in the HTML page that will serve as a test suite. In the real world, it's more likely that tax.js would contain a number of different functions, perhaps as part of a financial application, but for this example it'll do just fine. The HTML code is shown in Listing 24.1.

LISTING 24.1 HTML Code for the Test Suite

Click here to view code image

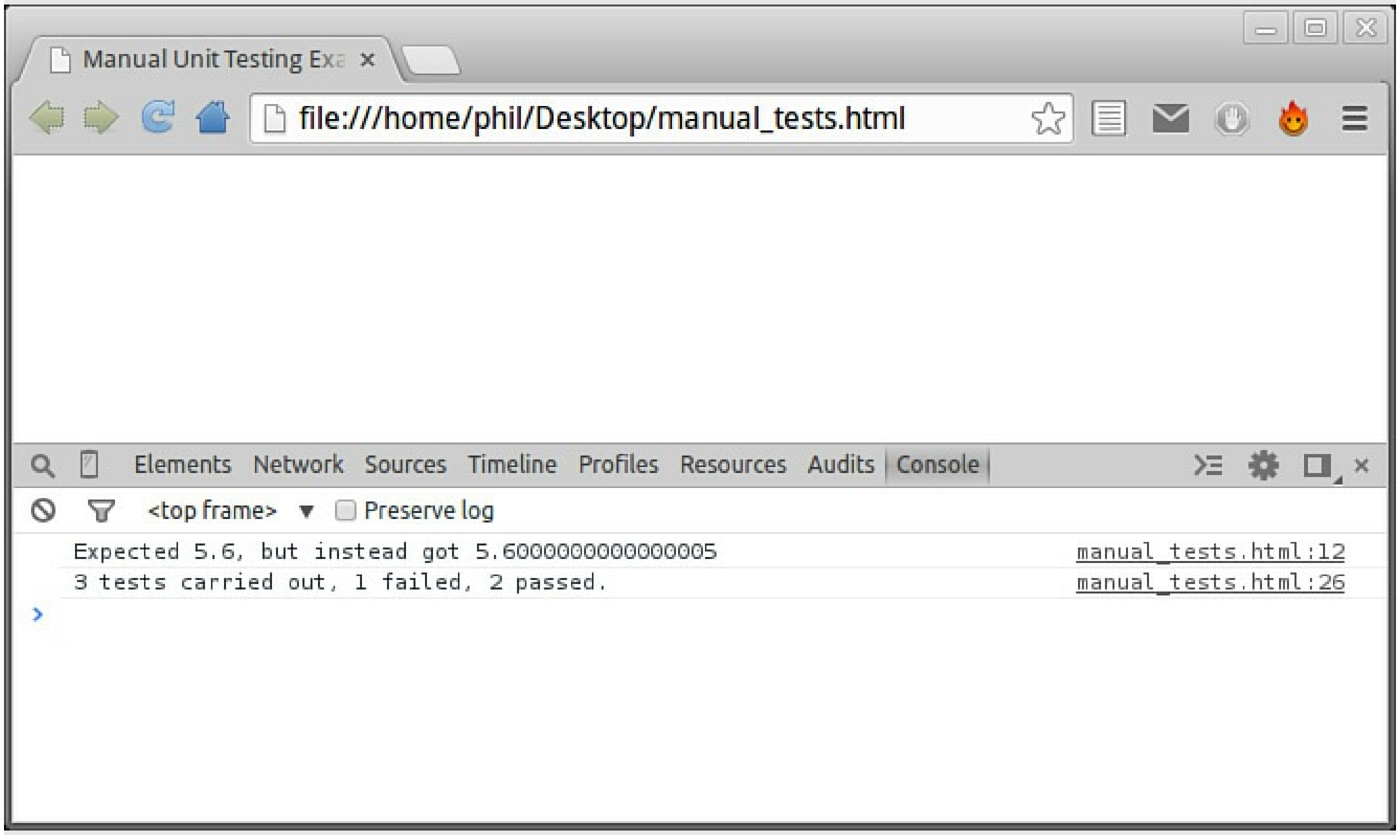
```
<!DOCTYPE html>
<html>
<head>
  <title>Manual Unit Testing Examples</title>
  <script src="tax.js"></script>
  <script>
  function test(amount, rate, expected) {
    results.total++;
    var result = addTax(amount, rate);
    if (result !== expected) {
      results.failed++;
      console.log("Expected " + expected + ", but instead got " + result);
    }
  }
  var results = {
    total: 0,
    failed: 0
  };

  // Our unit tests
  test(1, 10, 1.1);
  test(5, 12, 5.6);
  test(100, 17.5, 117.5);
```

```
    // Output results to the console
    console.log(results.total + " tests carried out, " + results.failed + "
  failed, " +(results.total - results.failed) + " passed.");
    </script>
  </head>
  <body>
  </body>
  </html>
```

In a script element on this page I've defined a function called `test()`, which takes three arguments. The parameters `amount` and `rate` are the values to be passed to the function under test (in this case `addTax()`) while the third parameter, `expected`, is the result we expect to have returned if the `addTax()` function is working the way we want it to.

If the function under test returns the expected result, the `test()` function increments a counter of successful tests; if not, it increments a counter of failed tests and also logs a message to the console indicating which test failed, and the incorrect value that was returned.

To run the test I can simply save the code as an .html file and load this page into my browser. The result is shown in Figure 24.1.


Manual Unit Testing Exa
file:///home/phil/Desktop/manual_tests.html
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
Preserve log

```
Expected 5.6, but instead got 5.6000000000000005          manual_tests.html:12
3 tests carried out, 1 failed, 2 passed.                  manual_tests.html:26
```

FIGURE 24.1 The results of our unit tests

As you can see, one of our tests failed; JavaScript added a small rounding error. This result is a starting point from which I can reexamine the code of the function `addTax()` to remove this anomaly—perhaps by using rounding on the output value before returning it.

Of course, you don't have to log the results to the console; they could just as easily be printed to the page, saved to a database, and so on. Later we'll look at QUnit, an open source unit testing application that displays a nicely formatted test result directly in the HTML of the test page.

Writing JavaScript for Unit Testing

Looking again at the preceding example, some things immediately become clear:

- The function `addTax()` was, of course, a *named* function. Had it been an anonymous function, it would have been much harder—perhaps impossible—to test this way.
- The function having been contained in an external JavaScript file (here tax.js) also aided in its inclusion in our test suite.

These are just a couple of examples of how coding style can affect the testability of your JavaScript code.

Applications written in the old-fashioned procedural manner of coding can be hard to unit test. However, if you write your code with unit testing in mind, you'll not only make it easier to write those tests, but also write code that's so much easier to read, maintain, and extend.

In the next section I'll round up a few coding techniques that will make your code easier to test.

Refactoring Code

The process of reorganizing your JavaScript code into a different form, but without modifying its operation, is known as *refactoring*. Refactoring is a great method of improving the design of a program, as it generally involves separating the program logic from the user interaction and display elements.

Where your code is based around a library such as jQuery and/or a framework such as AngularJS, some of that discipline might already have been imposed on your code. Where JavaScript has been written from scratch, however, there's more likelihood that a lot of refactoring will be needed. This is especially true where code has been initially written with little or no concern for separating HTML and program logic—for instance, where inline event handlers have been used.

Here are a few of the things you can do to make your code easier to test.

Externalize Scripts

JavaScript code collected into external files and later linked into your application is generally easier to test. In an ideal case, the same JavaScript file to be included in your production application can simply be linked into your test harness instead in order to have the tests performed.

Keep Functions and Methods Simple

The more you try to do with a single procedure, the more complex and numerous your tests will have to be, and the more difficult it may be to untangle that function from the code around it in order to test it.

The QUnit Test Suite

The previous example shows that you can run practical unit tests without a huge amount of code. However, for more than just a few tests it's much more productive to use a purpose-designed unit testing framework providing more advanced tools for writing and running your tests. In this section we'll look at one such framework, *QUnit*.

QUnit is a powerful, open source JavaScript unit testing framework. It's written by members of the jQuery team, and is used in the jQuery, jQuery UI, and jQuery Mobile projects. However, QUnit is capable of testing any regular JavaScript code.

Installing QUnit

QUnit is a self-contained library, needing only one JavaScript file (`qunit.js`) and one CSS file (`qunit.css`), both of which you can download from the QUnit website. Alternatively, you can use versions that are hosted on a CDN, as in the following examples.

A Minimal QUnit Setup

Once again, our test setup will be a simple HTML page, as shown in Listing 24.2. This time, QUnit is included from a CDN.

LISTING 24.2 Code for QUnit Test Suite

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
<title>Hello QUnit Example</title>
<link rel="stylesheet" href="http://code.jquery.com/qunit/qunit-
1.16.0.css">
</head>
<body>
```

```
<div id="qunit"></div>
<div id="qunit-fixture"></div>
<script src="http://code.jquery.com/qunit/qunit-1.16.0.js"></script>
<script src="tests.js"></script>
</body>
</html>
```

Note the layout of this file, and how similar it is to our manual test suite used in the previous section. There are a few differences, however:

- Instead of logging information to the console, QUnit outputs nicely formatted results into the `<div>` having ID `#qunit`.
- The `#qunit-fixture` element is required in instances where you need to set up a so-called *mock DOM* of elements that are used during testing, usually things like form elements. The mock DOM is reset after every unit test. QUnit expects the mock DOM to be in the `#qunit-fixture` element.

We include the CSS file in the head, while the body includes the QUnit JavaScript file, followed by a further file called tests.js that contains just the following code:

Click here to view code image

```
QUnit.test( "Hello QUnit test", function( assert ) {
assert.ok( 1 == "1", "Passed!" );
});
```

Let's see what's happening here. The test method is called, giving a name to the test as the first argument, and passing a function as the second argument. It's this function that will run our test.

Click here to view code image

```
assert.ok( 1 == "1", "Passed!" );
```

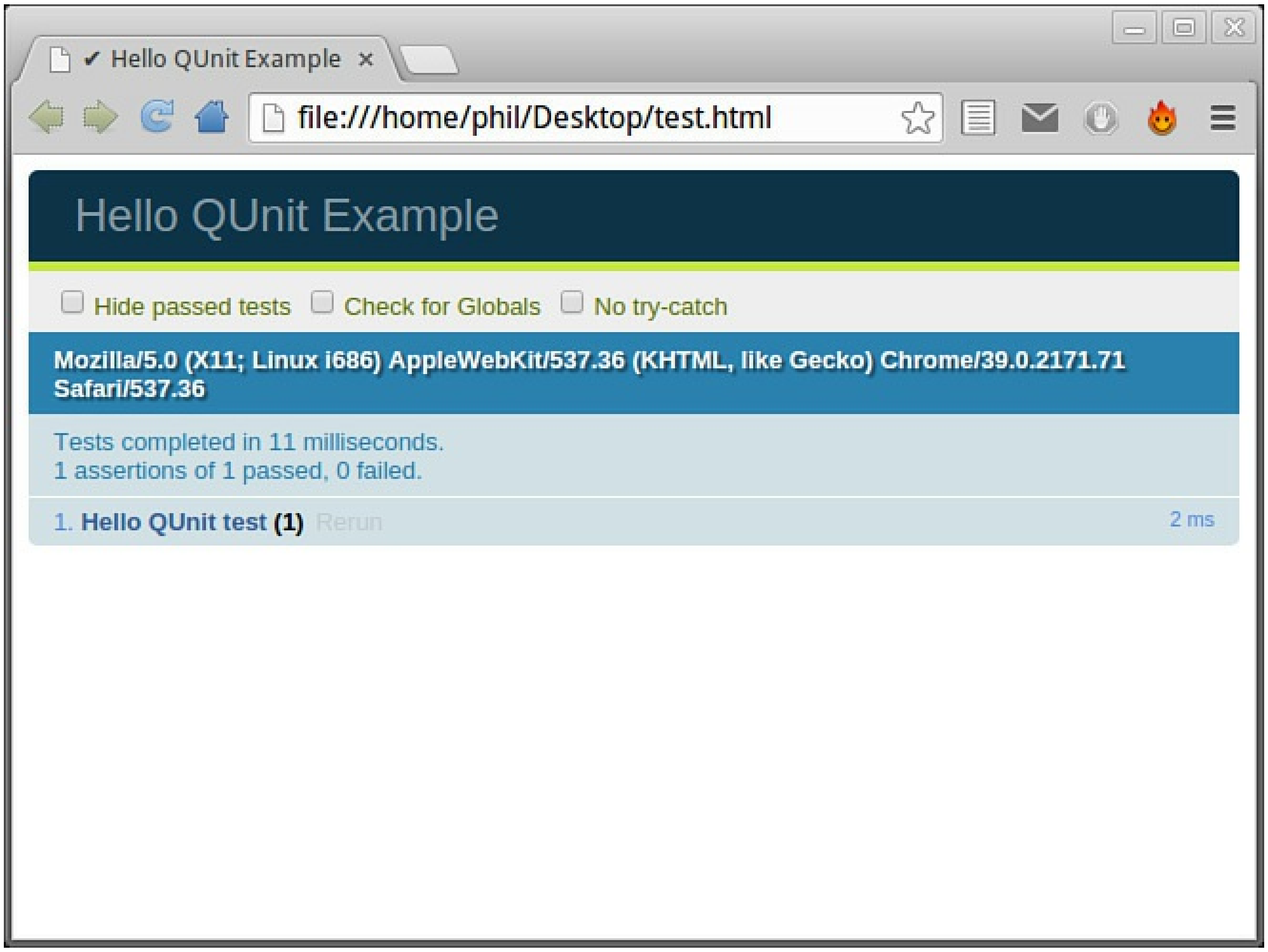
In this trivial example the test will always be passed. The `ok` method is one of several *assertions* that QUnit provides, and returns a value of true if the first parameter passed to it returns something that itself evaluates to true.

QUnit Assertions

An *assertion* evaluates expressions to true or false. QUnit's `assert` has a number of additional methods you can use to build your tests. Examples include `equal()`, which unsurprisingly, tests equality; `notEqual()`, which tests for inequality; and `strictEqual()`, which once again tests equality, but this time with a strict type and value comparison.

Check out all of the assertions in QUnit's documentation at http://api.qunitjs.com/category/assert/.

The result is shown in Figure 24.2.



Hello QUnit Example

Hide passed tests Check for Globals No try-catch

Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36

Tests completed in 11 milliseconds.
1 assertions of 1 passed, 0 failed.

1. Hello QUnit test (1) Rerun 2 ms

FIGURE 24.2 The result of our sample unit test

Retesting Our `addTax()` Function

Now let's modify the preceding example to retest our function `addTax()`, this time using QUnit. The test harness is similar to before, except it now includes the file tax.js:

Click here to view code image

```
<!DOCTYPE html>
<html>
<head>
<title>Test of addTax Function with QUnit</title>
<link rel="stylesheet" href="http://code.jquery.com/qunit/qunit-
1.16.0.css">
<script src="tax.js"></script>
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fixture"></div>
<script src="http://code.jquery.com/qunit/qunit-1.16.0.js"></script>
<script src="tests.js"></script>
</body>
```

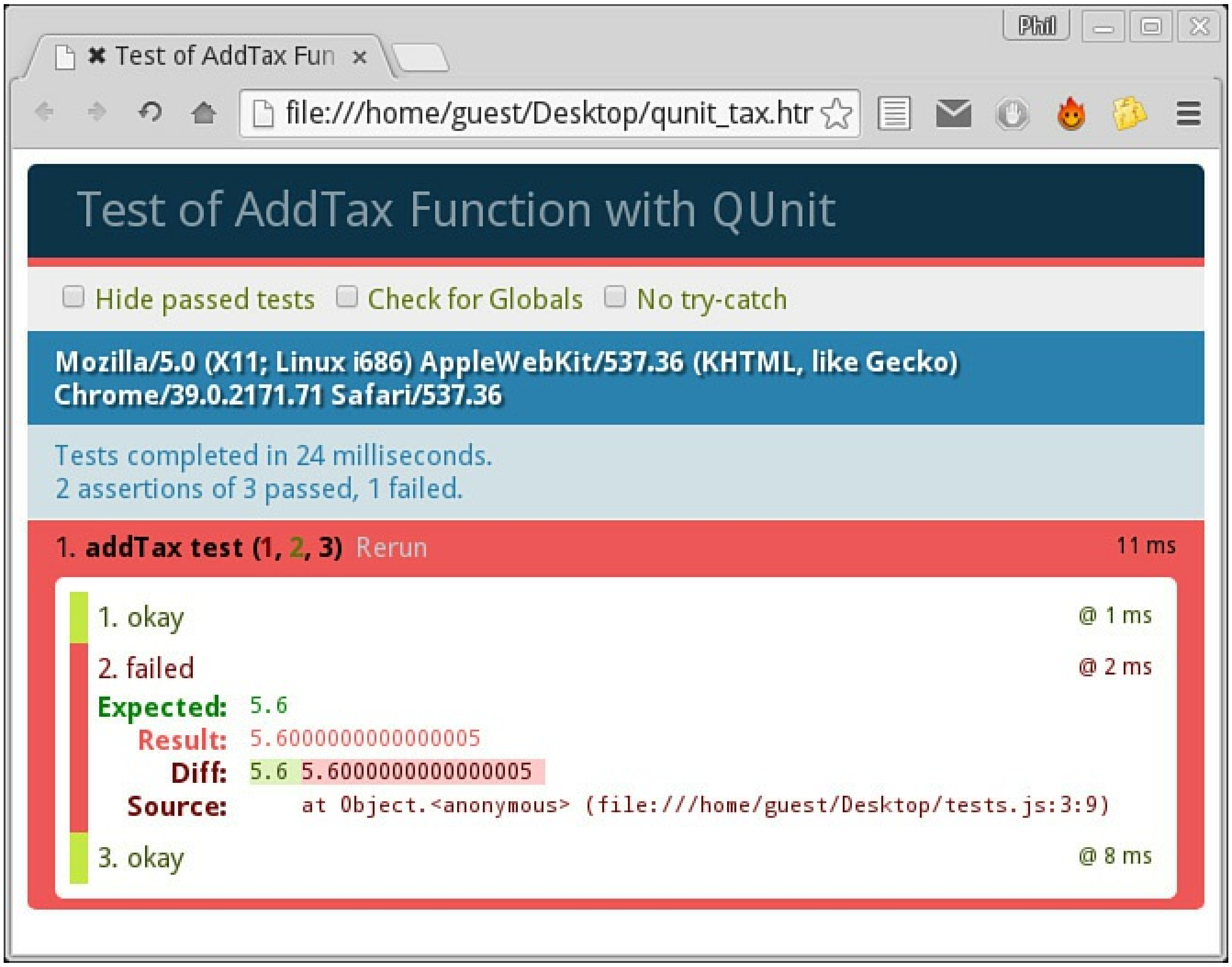
```
    </html>
```

Now we need to modify the file tests.js to include the tests we want to run. Here, I've used a different assertion, `assert.equal`, which is passed if the two parameters offered to it are equal. The first parameter in each test is the value returned from the `addTax()` function being tested, while the second parameter is our expected result:

Click here to view code image

```
    QUnit.test( "addTax test", function( assert ) {
        assert.equal(addTax(1, 10), 1.1);
        assert.equal(addTax(5, 12), 5.6);
        assert.equal(addTax(100, 17.5), 117.5);
    });
```

When the test harness is loaded into a browser, the output is as shown in Figure 24.3.



Test of AddTax Function with QUnit

Hide passed tests   Check for Globals   No try-catch

Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.71 Safari/537.36

Tests completed in 24 milliseconds.
2 assertions of 3 passed, 1 failed.

1. addTax test (1, 2, 3) Rerun 11 ms

1. okay @ 1 ms

2. failed @ 2 ms

```
Expected: 5.6
  Result: 5.6000000000000005
    Diff: 5.6 5.6000000000000005
  Source:     at Object.<anonymous> (file:///home/guest/Desktop/tests.js:3:9)
```

3. okay @ 8 ms

FIGURE 24.3 Testing our `addTax` function with QUnit

You'll notice that the formatting is much more user-friendly than our previous home-cooked version, but the result is identical; one test of the three failed.

Summary

In this hour, you learned some methods for unit testing your JavaScript code.

Testing JavaScript often requires some important changes in the structure of your code, and you read about some of the coding techniques you might employ to make code testing easier.

You saw how to run some tests with a home-cooked testing framework, then replaced that with the purpose-designed QUnit framework to carry out further testing.

Q&A

Q. Is there a common standard for unit testing?

A. Although there is no universal standard, the CommonJS project (http://en.wikipedia.org/wiki/CommonJS) has a unit testing specification (http://wiki.commonjs.org/wiki/Unit_Testing), which is the one utilized by the QUnit test suite described in this hour.

Q. Where did QUnit come from?

A. Originally, QUnit was developed by John Resig as part of the jQuery library. In 2008 it became a self-contained project, allowing non-jQuery users to use it for their own unit testing. A rewrite in 2009 removed its dependence on jQuery, and now QUnit runs standalone.

Workshop

Try to answer all the questions before reading the subsequent "Answers" section.

Quiz

1. In unit testing, the term *unit* refers to:

   a. The smallest testable part of an application, such as an individual function or method

   b. All of the JavaScript code active on a single web page

   c. All of the JavaScript code in a .js file included in your web page

2. An assertion:

   a. Always declares a test experssion to be true

   b. Evaluates a test expression as being true or false

   c. Sets the value of a JavaScript variable

3. Reworking your JavaScript code to make it more testable is usually referred to as:

   a. Restructuring your code

b. Refactoring your code

c. Recompiling your code

Answers

1. a. The smallest testable part of an application, such as an individual function or method
2. b. Evaluates an expression as being true or false
3. b. Refactoring your code

Exercises

- Modify the "home cooked" unit testing code to display a neatly formatted report into the HTML page instead of logging it to the console (as QUnit does).
- Pick some other functions from throughout the book and subject them to unit tests using QUnit. Do they all work as you expected?

Part IX: Appendices

Appendix A. Tools for JavaScript Development

JavaScript development doesn't require any special tools or software other than a text editor and a browser.

Most operating systems come bundled with at least one of each, and in many cases these tools will be more than sufficient for you to write your code.

However, many alternative and additional tools are available, some of which are described here.

Tip

Be sure to check the license terms on the individual websites and/or included in the download package.

Editors

The choice of an editor program is a personal thing, and most programmers have their favorite. Listed in the following sections are some popular, free editors that you can try.

Notepad++

If you develop on the Windows platform, you're probably already aware of the Notepad editor usually bundled with Windows. Notepad++ (http://notepad-plus-plus.org/) is a free application that aims to be a more powerful replacement, while still being light and fast.

Notepad++ offers line numbering, syntax and brace highlighting, macros, search and replace, and a whole lot more.

jEdit

jEdit is a free editor written in Java. It can therefore be installed on any platform having a Java virtual machine available, such as Windows, Mac OS X, OS/2, Linux, and so on.

A fully featured editor in its own right, jEdit can also be extended via 200+ available plug-ins to become, for example, a complete development environment or an advanced XML/HTML editor.

Download jEdit from www.jedit.org.

SciTE

Initially developed as a demonstrator for the Scintilla editing component, SciTE has developed into a complete and useful editor in its own right.

A free version of SciTE is available for Windows and Linux users via download from www.scintilla.org/SciTE.html, while a commercial version is available via the Mac Apps Store for Mac OS X users.

Geany

Geany (www.geany.org/) is a capable editor that can also be used as a basic integrated development environment (IDE). It was developed to provide a small and fast IDE, and can be installed on pretty much any platform supported by the GTK toolkit, including Windows, Linux, Mac OS X, and FreeBSD.

Geany is free to download and use under the terms of the GNU General Public License.

Validators

To make sure your pages work as intended regardless of the user's browser and operating system, it's always advisable to check your HTML code for correctness and conformance to standards.

A number of online tools and facilities are available to help you, as discussed next.

The W3C Validation Services

The W3C offers an online validator at http://validator.w3.org/ that will check the markup validity of web documents in HTML, XHTML, SMIL, MathML, and other markup languages. You can enter the URL of the page to be checked, or cut-and-paste your code directly into the validator.

CSS can be validated in a similar way at http://jigsaw.w3.org/css-validator/validator.html.en.

Web Design Group (WDG)

WDG also offers an online validation service at www.htmlhelp.com/tools/validator/.

This is similar to the W3C validator, but in some circumstances gives slightly more helpful information, such as warnings about valid but dangerous code, or highlighting undefined references rather than simply listing them as errors.

Debugging and Verifying Tools

Debugging tools can save you hours when trying to track down elusive problems in your JavaScript code and help you speed up your scripts by analyzing execution timing.

Verifying tools help you to write tidy, concise, readable, and problem-free code.

Numerous debugging and verifying tools are available, including the following.

Firebug

Firebug integrates with the Mozilla Firefox browser to offer excellent debugging, editing, and profiling tools. Go to http://getfirebug.com/javascript.

JSLint

JSLint (http://www.jslint.com/), written by Douglas Crockford, analyzes your JavaScript source code and reports potential problems, including both style conventions and coding errors.

Appendix B. JavaScript Quick Reference

Table B.1, Table B.2, Table B.3, and Table B.4 in this appendix contain a quick look-up for some of the more commonly used elements of JavaScript syntax, along with properties and methods for a selection of the built-in objects.

Operator	Description
Arithmetic Operators	
*	Multiplies two numbers.
/	Divides two numbers.
% (Modulus)	Returns the remainder left after dividing two numbers using integer division.
String Operators	
+	(String addition) Joins two strings.
+=	Joins two strings and assigns the joined string to the first operand.
Logical Operators	
&&	(Logical AND) Returns a value of true if both operands are true; otherwise, returns false.
||	(Logical OR) Returns a value of true if either operand is true. However, if both operands are false, returns false.
!	(Logical NOT) Returns a value of false if its operand is true; true if its operand is false.
Bitwise Operators	
&	(Bitwise AND) Returns a one in each bit position if both operands' bits are one.
^	(Bitwise XOR) Returns a one in a bit position if the bits of one operand, but not both operands, are one.
|	(Bitwise OR) Returns a one in a bit if either operand has a one in that position.
~	(Bitwise NOT) Changes ones to zeros and zeros to ones in all bit positions—that is, flips each bit.
<<	(Left shift) Shifts the bits of its first operand to the left by the number of places given in the second operand.

`>>`	(Sign-propagating right shift) Shifts the bits of the first operand to the right by the number of places given in the second operand.
`>>>`	(Zero-fill right shift) Shifts the bits of the first operand to the right by the number of places given in the second operand, and then shifts in zeros from the left.

Assignment Operators

`=`	Assigns the value of the second operand to the first operand, if the first operand is a variable.
`+=`	Adds two operands and assigns the result to the first operand, if it is a variable.
`-=`	Subtracts two operands and assigns the result to the first operand, if it is a variable.
`*=`	Multiplies two operands and assigns the result to the first operand, if it is a variable.
`/=`	Divides two operands and assigns the result to the first operand, if it is a variable.
`%=`	Calculates the modulus of two operands and assigns the result to the first operand, if it is a variable.
`&=`	Executes a bitwise AND operation on two operands and assigns the result to the first operand, if it is a variable.
`^=`	Executes a bitwise exclusive OR operation on two operands and assigns the result to the first operand, if it is a variable.
`|=`	Executes a bitwise OR operation on two operands and assigns the result to the first operand, if it is a variable.
`<<=`	Executes a left shift operation on two operands and assigns the result to the first operand, if it is a variable.
`>>=`	Executes a sign-propagating right shift operation on two operands and assigns the result to the first operand, if it is a variable.
`>>>=`	Executes a zero-fill right shift operation on two operands and assigns the result to the first operand, if it is a variable.

Comparison Operators	
`==`	(Equality operator) Returns true if the two operands are equal to each other.
`!=`	(Not-equal-to) Returns true if the two operands are not equal to each other.
`===`	(Strict equality) Returns true if the two operands are both equal and of the same type.
`!==`	(Strict not-equal-to) Returns true if the two operands are either not equal and/or not of the same type.
`>`	(Greater-than) Returns true if the first operand's value is greater than the second operand's value.
`>=`	(Greater-than-or-equal-to) Returns true if the first operand's value is greater than or equal to the second operand's value.
`<`	(Less-than) Returns true if the first operand's value is less than the second operand's value.
`<=`	(Less-than-or-equal-to) Returns true if the first operand's value is less than or equal to the second operand's value.
Special Operators	
`?:`	(Conditional operator) Executes an "if...else" test.
`,`	(Comma operator) Evaluates two expressions and returns the result of evaluating the second expression.
`delete`	(Deletion) Deletes an object and removes it from memory, or deletes an object's property, or deletes an element in an array.
`function`	Creates an anonymous function.
`in`	Returns true if the property you're testing is supported by a specific object.
`instanceof`	Returns true if the given object is an instance of the specified type.
`new`	Creates a new object from the specified object type.
`typeof`	Returns the name of the type of the operand.
`void`	Allows evaluation of an expression without returning a value.

TABLE B.1 The JavaScript Operators

Method	Description
`substring`	Returns a portion of the string.
`toUpperCase`	Converts all characters in the string to uppercase.
`toLowerCase`	Converts all characters in the string to lowercase.
`indexOf`	Finds an occurrence of a string within the string.
`lastIndexOf`	Finds an occurrence of a string within the string, starting at the end of the string.
`replace`	Searches for a match between a substring and a string, and replaces the substring with a new substring.
`split`	Splits a string into an array of substrings, and returns the new array.
`link`	Creates an HTML link using the string's text.
`anchor`	Creates an HTML anchor within the current page.

TABLE B.2 String Methods

Property	Description
Constants	
`E`	Base of natural logarithms (approximately 2.718).
`LN2`	Natural logarithm of 2 (approximately 0.693).
`LN10`	Natural logarithm of 10 (approximately 2.302).
`LOG2E`	Base 2 logarithm of *e* (approximately 1.442).
`LOG10E`	Base 10 logarithm of *e* (approximately 0.434).
`PI`	Ratio of a circle's circumference to its diameter (approximately 3.14159).
`SQRT1_2`	Square root of one half (approximately 0.707).
`SQRT2`	Square root of two (approximately 1.4142).
Method	**Description**
Algebraic	
`acos`	Arc cosine of a number in radians.
`asin`	Arc sine of a number.
`atan`	Arc tangent of a number.
`cos`	Cosine of a number.
`sin`	Sine of a number.
`tan`	Tangent of a number.

Statistical and Logarithmic	
`exp`	Returns ***e*** (the base of natural logarithms) raised to a power.
`log`	Returns the natural logarithm of a number.
`max`	Accepts two numbers and returns whichever is greater.
`min`	Accepts two numbers and returns the smaller of the two.
Property	**Description**
Basic and Rounding	
`abs`	Absolute value of a number.
`ceil`	Rounds a number up to the nearest integer.
`floor`	Rounds a number down to the nearest integer.
`pow`	One number to the power of another.
`round`	Rounds a number to the nearest integer.
`sqrt`	Square root of a number.
Random Numbers	
`random`	Random number between 0 and 1.

TABLE B.3 The Math Object

Method	Description
`getDate()`	Returns day of the month (1-31).
`getDay()`	Returns day of the week (0-6).
`getFullYear()`	Returns year (four digits).
`getHours()`	Returns hour (0-23).
`getMilliseconds()`	Returns milliseconds (0-999).
`getMinutes()`	Returns minutes (0-59).
`getMonth()`	Returns month (0-11).
`getSeconds()`	Returns seconds (0-59).
`getTime()`	Returns number of milliseconds since midnight Jan 1, 1970.
`getTimezoneOffset()`	Returns time difference between GMT and local time, in minutes.
`getUTCDate()`	Returns day of the month, according to Universal Time (1-31).
`getUTCDay()`	Returns day of the week, according to Universal Time (0-6).
`getUTCFullYear()`	Returns year, according to Universal Time (4 digit).
`getUTCHours()`	Returns hour, according to Universal Time (0-23).
`getUTCMilliseconds()`	Returns milliseconds, according to Universal Time (0-999).
`getUTCMinutes()`	Returns minutes, according to Universal Time (0-59).
`getUTCMonth()`	Returns month, according to Universal Time (0-11).

`getUTCSeconds()`	Returns seconds, according to Universal Time (0-59).
`parse()`	Parses a date string and Returns number of milliseconds since midnight of January 1, 1970.
`setDate()`	Sets the day of the month (1-31).
`setFullYear()`	Sets the year (four digits).
`setHours()`	Sets the hour (0-23).
`setMilliseconds()`	Sets the milliseconds (0-999).
`setMinutes()`	Set the minutes (0-59).
`setMonth()`	Sets the month (0-11).
`setSeconds()`	Sets the seconds (0-59).
`setTime()`	Sets a date and time by adding or subtracting a specified number of milliseconds to or from midnight January 1, 1970.
`setUTCDate()`	Sets the day of the month, according to Universal Time (1-31).
`setUTCFullYear()`	Sets the year, according to Universal Time (four digits).
`setUTCHours()`	Sets the hour, according to Universal Time (0-23).
`setUTCMilliseconds()`	Sets the milliseconds, according to Universal Time (0-999).
`setUTCMinutes()`	Set the minutes, according to Universal Time (0-59).
`setUTCMonth()`	Sets the month, according to Universal Time (0-11).
`setUTCSeconds()`	Set the seconds, according to Universal Time (0-59).
`toDateString()`	Converts the date part of a `Date` object into a readable string.
`toLocaleDateString()`	Returns the date part of a `Date` object as a string, using locale conventions.
`toLocaleTimeString()`	Returns the time part of a `Date` object as a string, using locale conventions.
`toLocaleString()`	Converts a `Date` object to a string, using locale conventions.
`toString()`	Converts a `Date` object to a string.
`toTimeString()`	Converts the time part of a `Date` object to a string.
`toUTCString()`	Converts a `Date` object to a string, according to Universal Time.
`UTC()`	Returns the number of milliseconds in a date string since midnight of January 1, 1970, according to Universal Time.
`valueOf()`	Returns the primitive value of a `Date` object.

TABLE B.4 The Date Object

Index

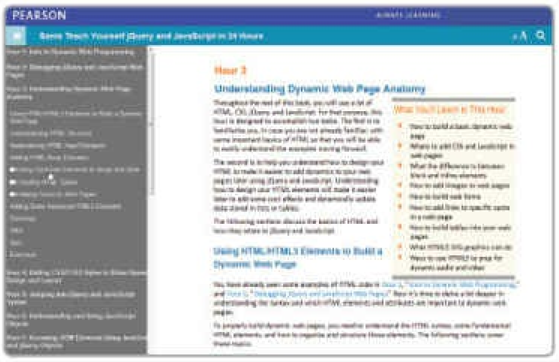
Learning Labs!

Learn online with videos, live code editing, and quizzes

SPECIAL 50% OFF – Introductory Offer
Discount Code: STYLL50

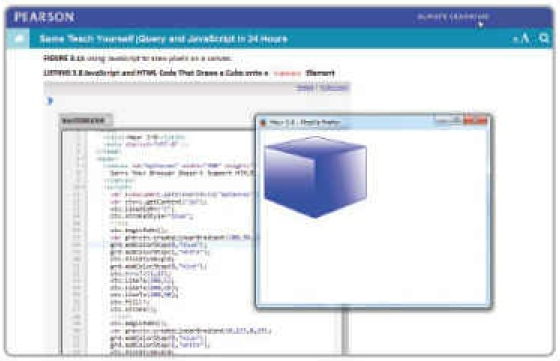
FOR A LIMITED TIME, we are offering readers of **Sams Teach Yourself** books a **50% OFF** discount to **ANY online Learning Lab** through Dec 31, 2016.

Visit **informit.com/learninglabs** to see available labs, try out full samples, and order today.



- **Read** the complete text of the book online in your web browser

- **Watch** an expert instructor show you how to perform tasks in easy-to-follow videos



- **Try** your hand at coding in an interactive code-editing sandbox in select products

Given the HTML markup <p align="center">, what is the component that is represented by "align"?

- **Test** yourself with interactive quizzes

ALWAYS LEARNING PEARSON

Code Snippets

```
<script>
    ... JavaScript statements ...
</script>
```

```
<script type="text/javascript">
    ... JavaScript statements ...
</script>
```

```
<script>window.alert("Here is my message");</script>
```

```
<script>alert("Here is my message");</script>
```

```
<script>document.write("Here is another message");</script>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello from JavaScript!</title>
</head>
<body>
    <script>
        alert("Hello World!");
    </script>
</body>
</html>
```

```
<script>
    ... Javascript statements are written here ...
</script>
```

```
<script src='mycode.js'></script>
```

```
<script src='/path/to/mycode.js'></script>
```

```
<script src='http://www.example.com/path/to/mycode.js'></script>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>A Simple Page</title>
</head>
<body>
    <p>Some content ...</p>
    <script src='mycode.js'></script>
</body>
</html>
```

```
<script>
    <!--
    ... Javascript statements are written here ...
    -->
</script>
```

```
this is statement 1; this is statement 2;
```

```
var productName = "Leather wallet";
var productCount = 2;
var subtotal = 14.98;
var shipping = 2.75;
var total = subtotal + shipping;
```

```
subtotal = total - shipping;
var salesTax = total * 0.15;
var productPrice = subtotal / productCount;
var itemsPerBox = 12;
var itemsToBeBoxed = 40;
var itemsInLastBox = itemsToBeBoxed % itemsPerBox;
```

```
productCount = productCount + 1;
```

```
var average = (a + b + c) / 3;
var firstname = "John";
var surname = "Doe";
var fullname = firstname + " " + surname;
// the variable fullname now contains the value "John Doe"
```

```
var cTemp = 100;  // temperature in Celsius
// Let's be generous with parentheses
var hTemp = ((cTemp * 9) /5 ) + 32;
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Fahrenheit From Celsius</title>
</head>
<body>
    <script>
        var cTemp = 100;  // temperature in Celsius
        // Let's be generous with parentheses
        var hTemp = ((cTemp * 9) /5 ) + 32;
        document.write("Temperature in Celsius: " + cTemp + " degrees<br/>");
        document.write("Temperature in Fahrenheit: " + hTemp + " degrees");
    </script>
</body>
</html>
```

```
onclick=" ...some JavaScript code... "
```

```
<!DOCTYPE html>
<html>
<head>
    <title>onClick Demo</title>
</head>
<body>
    <input type="button" onclick="alert('You clicked the button!')" value="Click
    ➥Me" />
</body>
</html>
```

```
alert('You clicked the button!')
```

```
<!DOCTYPE html>
<html>
<head>
    <title>onMouseOver Demo</title>
</head>
<body>
    <img src="image1.png" alt="image 1" onmouseover="alert('You entered the
    ➥image!')" />
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>OnMouseOver Demo</title>
</head>
<body>
    <img src="tick.gif" alt="tick" onmouseover="this.src='tick2.gif';"
onmouseout="this.src='tick.gif';" />
</body>
</html>
```

```
function sayHello() {
    alert("Hello");
    // ... more statements can go here ...
}
```

```
<input type="button" value="Say Hello" onclick="sayHello()" />
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Calling Functions</title>
    <script>
        function sayHello() {
            alert("Hello");
        }
    </script>
</head>
<body>
    <input type="button" value="Say Hello" onclick="sayHello()" />
</body>
</html>
```

```
function buttonReport(buttonId, buttonName, buttonValue) {
    // information about the id of the button
    var userMessage1 = "Button id: " + buttonId + "\n";
    // then about the button name
    var userMessage2 = "Button name: " + buttonName + "\n";
    // and the button value
    var userMessage3 = "Button value: " + buttonValue;
    // alert the user
    alert(userMessage1 + userMessage2 + userMessage3);
}
```

```
<input type="button" id="id1" name="Button 1" value="Something" />
```

```
onclick = "buttonReport(this.id, this.name, this.value)"
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Calling Functions</title>
    <script>
        function buttonReport(buttonId, buttonName, buttonValue) {
            // information about the id of the button
            var userMessage1 = "Button id: " + buttonId + "\n";
            // then about the button name
            var userMessage2 = "Button name: " + buttonName + "\n";
            // and the button value
            var userMessage3 = "Button value: " + buttonValue;
            // alert the user
            alert(userMessage1 + userMessage2 + userMessage3);
        }
    </script>
</head>
<body>
    <input type="button" id="id1" name="Left Hand Button" value="Left" onclick
    ➥="buttonReport(this.id, this.name, this.value)"/>
    <input type="button" id="id2" name="Center Button" value="Center" onclick
    ➥="buttonReport(this.id, this.name, this.value)"/>
    <input type="button" id="id3" name="Right Hand Button" value="Right" onclick
    ➥="buttonReport(this.id, this.name, this.value)"/>
</body>
</html>
```

```
// Define our function addTax()
function addTax(subtotal, taxRate) {
    var total = subtotal * (1 + (taxRate/100));
    return total;
}
// now let's call the function
var invoiceValue = addTax(50, 10);
alert(invoiceValue); // works correctly
alert(total);  // doesn't work
```

```
var a = 10;
var b = 10;
function showVars() {
    var a = 20; // declare a new local variable 'a'
    b = 20;     // change the value of global variable 'b'
    return "Local variable 'a' = " + a + "\nGlobal variable 'b' = " + b;
}
var message = showVars();
alert(message + "\nGlobal variable 'a' = " + a);
```

```
{
    "name": "My First Extension",
    "version": "1.0",
    "manifest_version": 2,
    "description": "Hello World extension.",
    "browser_action": {
        "default_icon": "icon.png",
        "default_popup": "popup.html"
    },
    "web_accessible_resources": [
        "icon.png",
        "popup.js"
    ]
}
```

```
<!DOCTYPE html>
<html>
<head>
    <style>
        body {
            width: 350px;
        }
        div {
            border: 1px solid black;
            padding: 20px;
            font: 20px normal helvetica, verdana, sans-serif;
        }
    </style>
    <script src="popup.js"></script>
</head>
<body>
</body>
</html>
```

```
function sayHello() {
    var message = document.createTextNode("Hello World!");
    var out = document.createElement("div");
    out.appendChild(message);
    document.body.appendChild(out);
}
window.onload = sayHello;
```

{"name":"San Francisco
International","ICAO":"KSFO","state":"California","status":{"avgDelay":"",
"closureEnd":"", "closureBegin":"","type":"","minDelay":"","trend":"",
"reason":"No known delays for this airport.","maxDelay":"","endTime":""},
"delay":"false","IATA":"SFO","city":"San Francisco","weather":{"weather":"Partly
Cloudy",
"meta":{"credit":"NOAA's National Weather Service","url":"http://weather.gov/",
"updated":"1:56 AM Local"},"wind":"Southwest at 9.2mph","temp":"44.0 F (6.7 C)",
"visibility":"10.00"}}

```
{
    "name": "Airport Information",
    "version": "1.0",
    "manifest_version": 2,
    "description": "Information on US airports",
    "browser_action": {
        "default_icon": "plane.png",
        "default_popup": "popup.html"
    },
    "web_accessible_resources": [
    "plane.png",
    "popup.js"
  ],
    "permissions": [
    "http://services.faa.gov/"
  ]
}
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Airport Information</title>
    <style>
        body {
            width:350px;
            font: 12px normal arial, verdana, sans-serif;
        }
        #info {
            border: 1px solid black;
            padding: 10px;
        }
    </style>
</head>
<body>
    <h2>Airport Information</h2>
    <input type=Text id="airportCode" value="SFO" size="6" />
    <input id="btn" type="button" value="Get Information" />
    <div id="info"></div>
</body>
</html>
```

```
<script src="jquery-1.11.2.min.js" /></script>
```

```
$(document).ready(function(){
   $("#btn").click(function(){
      $("#info").html("Getting information ...");
      var code = $("#airportCode").val();
      $.get("http://services.faa.gov/airport/status/" + code +
      ➥"?format=application/json",
         '',
         function(data){
            displayData(data);
         }
      );
   });
});
```

```
$("#info").html("Getting information ...");
```

```
var code = $("#airportCode").val();
```

```
$.get("http://services.faa.gov/airport/status/" + code + "?format=application/
json",
    '',
    function(data){
        displayData(data);
    }
);
```

```
function displayData(data) {
   var message = "Airport: " + data.name + "<br />";
   message += "<h3>STATUS:</h3>";
   for (i in data.status) {
      if(data.status[i] != "") message += i + ": " + data.status[i] + "<br />";
   }
   message += "<h3>WEATHER:</h3>";
   for (i in data.weather) {
      if(i != "meta") message += i + ": " + data.weather[i] + "<br />";
   }
   $("#info").html(message);
}
$(document).ready(function(){
   $("#btn").click(function(){
      $("#info").html("Getting information ...");
      var code = $("#airportCode").val();
      $.get("http://services.faa.gov/airport/status/" + code +
      ➥"?format=application/json",
         '',
         function(data){
            displayData(data);
         }
      );
   });
});
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Airport Information</title>
    <style>
        body {
            width:350px;
            font: 12px normal arial, verdana, sans-serif;
        }
        #info {
            border: 1px solid black;
            padding: 10px;
        }
    </style>
    <script src="jquery-1.11.2.min.js" /></script>
    <script src="popup.js"></script>
</head>
<body>
    <h2>Airport Information</h2>
    <input type=Text id="airportCode" value="SFO" size="6" />
    <input id="btn" type="button" value="Get Information" />
    <div id="info"></div>
</body>
</html>
```

```
function calculateGroundAngle(x1, y1, z1, x2, y2, z2) {
    /**
    * Calculates the angle in radians at which
    * a line between two points intersects the
    * ground plane.
    * @author Phil Ballard phil@www.example.com
    */
    if(x1 > 0) {
        .... more statements
```

```
// need to use our custom sort method for performance reasons
var finalArray = rapidSort(allNodes, byAngle) {
    .... more statements
```

```
// workaround for image onload bug in browser X version Y
if(!loaded(image1)) {
     .... more statements
```

```
// You can change the following dimensions to your preference:
var height = 400px;
var width = 600px;
```

```
function Car(make, model, color) {
    .... statements
}
```

```
var herbie = new Car('VW', 'Beetle', 'white');
```

```
function getElementArea() {
    var high = document.getElementById("id1").style.height;
    var wide = document.getElementById("id1").style.width;
    return high * wide;
}
```

```
function getElementArea(elementId) {
    var elem = document.getElementById(elementId);
    var high = elem.style.height;
    var wide = elem.style.width;
    return parseInt(high) * parseInt(wide);
}
```

```
var area1 = getElementArea("id1");
var area2 = getElementArea("id2");
```

```
function getElementArea(elementId) {
    if(document.getElementById(elementId)) {
        var elem = document.getElementById(elementId);
        var high = elem.style.height;
        var wide = elem.style.width;
        var area = parseInt(high) * parseInt(wide);
        if(!isNaN(area)) {
            return area;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

```
<input type="button" style="border: 1px solid blue;color: white"
onclick="doSomething()" />
```

```
<input type="button" class="blueButtons" onclick="doSomething()" />
```

```
<input type="button" class="blueButtons" id="btn1" />
```

```
function doSomething() {
     .... statements ....
}
document.getElementById("btn1").onclick = doSomething;
```

```
<form action="process.php">
<input id="searchTerm" name="term" type="text" /><br />
<input type="button" id="btn1" value="Search" />
</form>
```

```
function checkform() {
    if(document.forms[0].term.value == "") {
        alert("Please enter a search term.");
        return false;
    } else {
        document.forms[0].submit();
    }
}
window.onload = function() {
    document.getElementById("btn1").onclick = checkform;
}
```

```
<form action="process.php">
    <input id="searchTerm" name="term" type="text" /><br />
    <input type="submit" id="btn1" value="Search" />
</form>
```

```
function checkform() {
    if(document.forms[0].term.value == "") {
        alert("Please enter a search term.");
        return false;
    } else {
        return true;
    }
}
window.onload = function() {
    document.getElementById("btn1").onclick = checkform;
}
```

```
function setClipboard(myText){
    if((typeof clipboardData != 'undefined') && (clipboardData.setData)){
        clipboardData.setData("text", myText);
    } else {
        document.getElementById("copytext").innerHTML = myText;
        alert("Please copy the text from the 'Copy Text' field to your clipboard");
    }
}
```

```
if((typeof clipboardData != 'undefined') ....
... && (clipboardData.setData)){
```

```
document.getElementById("copytext").innerHTML = myText;
alert("Please copy the text from the 'copytext' field to your clipboard");
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Current Date and Time</title>
    <style>
        p {font: 14px normal arial, verdana, helvetica;}
    </style>
    <script>
        function telltime() {
            var out = "";
            var now = new Date();
            out += "<br />Date: " + now.getDate();
            out += "<br />Month: " + now.getMonth();
            out += "<br />Year: " + now.getFullYear();
            out += "<br />Hours: " + now.getHours();
            out += "<br />Minutes: " + now.getMinutes();
            out += "<br />Seconds: " + now.getSeconds();
            document.getElementById("div1").innerHTML = out;
        }
    </script>
</head>
<body>
    The current date and time are:<br/>
    <div id="div1"></div>
    <script>
        telltime();
    </script>
    <input type="button" onclick="location.reload()" value="Refresh" />
</body>
</html>
```

```
<noscript>
    Your browser does not support JavaScript<br />
    Please consult your computer's operating system for local date and
time information or click <a href="clock.php" target="_blank">HERE</a>
to read the server time.
</noscript>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Current Date and Time</title>
    <style>
        p {font: 14px normal arial, verdana, helvetica;}
    </style>
    <script src="datetime.js"></script>
</head>
<body>
    The current date and time are:<br/>
    <div id="div1"></div>
    <input id="btn1" type="button" value="Refresh" />
    <noscript>
        <p>Your browser does not support JavaScript.</p>
        <p>Please consult your computer's operating system for local date and time

information or click <a href="clock.php" target="_blank">HERE</a> to read the server

time.</p>
    </noscript>
</body>
</html>
```

```
function telltime() {
    var out = "";
    var now = new Date();
    out += "<br />Date: " + now.getDate();
    out += "<br />Month: " + now.getMonth();
    out += "<br />Year: " + now.getFullYear();
    out += "<br />Hours: " + now.getHours();
    out += "<br />Minutes: " + now.getMinutes();
    out += "<br />Seconds: " + now.getSeconds();
    document.getElementById("div1").innerHTML = out;
}

window.onload = function() {
    document.getElementById("btn1").onclick= function() {location.reload();}
    telltime();
}
```

```
<script>window.location="enhancedPage.html";</script>
```

```
function myFunc(a, b) {
   alert("myFunc() called.\na: " + a + "\nb: " + b);
   // .. rest of function code here ...
...}
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Strings and Arrays</title>
</head>
<body>
    <script>
        function sayHi() {
            alert("Hello!);
        }
    </script>
    <input type="button" value="good" onclick="sayHi()" />
    <input type="button" value="bad" onclick="sayhi()" />
</body>
</html>
```

```
<body onload="somefunction()" >
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Banner Cycler</title>
    <script>
        var banners = ["banner1.jpg", "banner2.jpg", "banner3.jpg"];
        var counter = 0;
        function run() {
            setInterval(cycle, 2000);
        }
        function cycle() {
            counter++;
            if(counter == banners.length) counter = 0;
            document.getElementById("banner").src = banners[counter];
        }
    </script>
</head>
<body onload = "run();">
    <img id="banner" alt="banner" src="banner1.jpg" />
</body>
</html>
```

```
if(counter == banners.length) counter = 0;
```

```
document.getElementById("banner").src = banners[counter];
```

```
function myFunc(a, b) {
   console.log("myFunc() called.\na: " + a + "\nb: " + b);
   // .. rest of function code here ...
...}
```

```
function myFunc(a, b) {
   console.group("myFunc execution");
   console.log("Executing myFunc()");
   if(isNaN(a) || isNaN(b)) {
      console.warn("One or more arguments non-numeric");
   }
   console.groupEnd();
   myOtherFunc(a+b);
}

function myOtherFunc(c) {
   console.group("myOtherFunc execution");
   console.log("Executing myOtherFunc()");
   if(isNaN(c)) {
      console.info("Argument is not numeric");
   }
   console.groupEnd();
   // .. rest of function code here ...
}
```

```
function myFunc(a, b) {
    if(isNaN(a) || isNaN(b)) {
    debugger;
    }
// .. rest of function code here ...
}
```

```
function addTax(subtotal, taxRate) {
    var total = subtotal * (1 + (taxRate/100));
    return total;
}
```

```
<!DOCTYPE html>
<html>
<head>
  <title>Manual Unit Testing Examples</title>
  <script src="tax.js"></script>
  <script>
  function test(amount, rate, expected) {
    results.total++;
    var result = addTax(amount, rate);
    if (result !== expected) {
      results.failed++;
      console.log("Expected " + expected + ", but instead got " + result);
    }
  }
  var results = {
    total: 0,
    failed: 0
  };

  // Our unit tests
  test(1, 10, 1.1);
  test(5, 12, 5.6);
  test(100, 17.5, 117.5);

  // Output results to the console
  console.log(results.total + " tests carried out, " + results.failed + " failed, "
  ➥+(results.total - results.failed) + " passed.");
  </script>
</head>
<body>
</body>
</html>
```

```html
<!DOCTYPE html>
<html>
<head>
<title>Hello QUnit Example</title>
<link rel="stylesheet" href="http://code.jquery.com/qunit/qunit-1.16.0.css">
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fixture"></div>
<script src="http://code.jquery.com/qunit/qunit-1.16.0.js"></script>
<script src="tests.js"></script>
</body>
</html>
```

```
QUnit.test( "Hello QUnit test", function( assert ) {
assert.ok( 1 == "1", "Passed!" );
});
```

```
assert.ok( 1 == "1", "Passed!" );
```

```html
<!DOCTYPE html>
<html>
<head>
<title>Test of addTax Function with QUnit</title>
<link rel="stylesheet" href="http://code.jquery.com/qunit/qunit-1.16.0.css">
<script src="tax.js"></script>
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fixture"></div>
<script src="http://code.jquery.com/qunit/qunit-1.16.0.js"></script>
<script src="tests.js"></script>
</body>
</html>
```

```
QUnit.test( "addTax test", function( assert ) {
    assert.equal(addTax(1, 10), 1.1);
    assert.equal(addTax(5, 12), 5.6);
    assert.equal(addTax(100, 17.5), 117.5);
}};
```