Modern JavaScript

DEVELOP AND DESIGN

Larry Ullman

Modern JavaScript

DEVELOP AND DESIGN

Larry Ullman

Peachpit Press

Modern JavaScript: Develop and Design

Larry Ullman

Peachpit Press
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)

Find us on the Web at: www.peachpit.com
To report errors, please send a note to: errata@peachpit.com
Peachpit Press is a division of Pearson Education.
Copyright © 2012 by Larry Ullman

Acquisitions Editor: Rebecca Gulick
Copy Editor: Patricia Pane
Technical Reviewer: Jacob Seidelin
Compositor: Danielle Foster
Production Editor: Katerina Malone
Proofreader: Liz Welch
Indexer: Valerie Haynes-Perry
Cover Design: Peachpit Press

Notice of Rights

All rights reserved. No part of this book may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. For information on getting permission for reprints and excerpts, contact permissions@peachpit.com.

Notice of Liability

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of the book, neither the author nor Peachpit Press shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademarks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Peachpit was aware of a trademark claim, the designations appear as requested by the owner of the trademark. All other product names and services identified throughout this book are used in editorial fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with this book.

13-digit ISBN: 978-0-321-81252-0
10-digit ISBN: 0-321-81252-2

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

This book is dedicated to Doug and Christina,
and to their family and friends,
for the extraordinary, life-changing gift.

SO MANY, MANY THANKS TO...

Rebecca, Nancy, and Nancy, for working very hard to make this project happen and for their supreme flexibility. And, of course, for continuing to work with me time and again.

Patricia, for her diligent editing and attention to detail.

Jacob, for providing a top-notch technical review, and for not being afraid to say "Not what I would do...."

Danielle, for magically converting a handful of random materials into something that looks remarkably like an actual book.

Liz, for the sharp proofreading eye. Never too late to catch a mistake!

The indexer, Valerie, who makes it easy for readers to find what they need without wading through all of my exposition.

Mimi, for the snazzy interior and cover design work. I love the tool motif!

All the readers over the years who requested that I write this book and provided detailed thoughts as to what they would and would not want this book to be. I hope it's what you were looking for!

Jonas Jacek (`http://jonas.me/`) for permission to use his HTML5 template.

Sara, for entertaining the kids so that I can get some work done, even if I'd rather not.

Sam and Zoe, for being the kid epitome of awesomeness.

Jessica, for doing everything you do and everything you can.

CONTENTS

Introduction . . . x
Welcome to JavaScript . . . xii

PART 1 GETTING STARTED

CHAPTER 1 **(RE-)INTRODUCING JAVASCRIPT . . . 2**
What Is JavaScript? . . . 4
JavaScript's History . . . 6
JavaScript Isn't... . . . 17
How JavaScript Compares to... . . . 18
Why JavaScript Is a Good Thing . . . 21
JavaScript Versions and Browser Support . . . 22
JavaScript Programming Goals . . . 24
Wrapping Up . . . 25

CHAPTER 2 **JAVASCRIPT IN ACTION . . . 26**
Choosing a Doctype . . . 28
An HTML5 Primer . . . 31
Adding JavaScript to HTML . . . 37
Key Development Approaches . . . 39
Cobbling Together Some Code . . . 44
Steal this JavaScript . . . 55
Wrapping Up . . . 56

CHAPTER 3 **TOOLS OF THE TRADE . . . 58**
The Great Debate: Text Editor or IDE? . . . 60
The Browser: Your Friend, Your Enemy . . . 69
Testing on Multiple Browsers . . . 75
Testing JavaScript . . . 77
Errors and Debugging . . . 80
Online Resources . . . 90
Wrapping Up . . . 91

PART 2 JAVASCRIPT FUNDAMENTALS

CHAPTER 4 **SIMPLE VARIABLE TYPES** 92

Basics of Variables 94

Working with Numbers 100

Working with Strings 112

Performing Type Conversions 122

Review and Pursue 125

Wrapping Up 127

CHAPTER 5 **USING CONTROL STRUCTURES** 128

Basics of Conditionals 130

More Conditionals 140

More Complex Conditions 153

Basics of Loops 161

Review and Pursue 168

Wrapping Up 169

CHAPTER 6 **COMPLEX VARIABLE TYPES** 170

Generating Dates and Times 172

Working with Arrays 190

Working with Objects 207

Arrays Versus Objects 216

Review and Pursue 217

Wrapping Up 219

CHAPTER 7 **CREATING FUNCTIONS** 220

The Fundamentals 222

Functions as Objects 244

The Fancier Stuff 254

Review and Pursue 263

Wrapping Up 265

CHAPTER 8 **EVENT HANDLING** 266

The Premise of Event Handling 268

Creating Event Listeners 268

Creating a Utility Library 275

Event Types 278
Event Accessibility 287
Events and Progressive Enhancement 288
Advanced Event Handling 290
Review and Pursue 305
Wrapping Up 307

CHAPTER 9 **JAVASCRIPT AND THE BROWSER 308**
Using Dialog Windows 310
Working with the Window 313
Manipulating the DOM 335
JavaScript and CSS 349
Working with Cookies 358
Using Timers 369
Review and Pursue 372
Wrapping Up 375

CHAPTER 10 **WORKING WITH FORMS 376**
General Form Considerations 378
Text Inputs and Textareas 387
Select Menus 389
Checkboxes 396
Radio Buttons 400
Handling File Uploads 401
Regular Expressions 403
Putting It All Together 415
Review and Pursue 421
Wrapping Up 423

CHAPTER 11 **AJAX 424**
Ajax Basics 426
Working with Other Data 442
The Server-Side Script 447
Ajax Examples 451
Review and Pursue 469
Wrapping Up 471

PART 3 NEXT STEPS

CHAPTER 12 **ERROR MANAGEMENT** **472**

Catching and Throwing Errors 474
Using Assertions 479
Unit Testing 481
Review and Pursue 488
Wrapping Up 489

CHAPTER 13 **FRAMEWORKS** **490**

Choosing a Framework 492
Introducing jQuery 494
Introducing YUI 509
Libraries 522
Review and Pursue 523
Wrapping Up 525

CHAPTER 14 **ADVANCED JAVASCRIPT** **526**

Defining Namespaces 528
Creating Custom Objects 529
Understanding Prototypes 537
Working with Closures 541
Alternative Type Identification 547
Minifying Code 548
Review and Pursue 550
Wrapping Up 551

CHAPTER 15 **PHP AND JAVASCRIPT TOGETHER** 552
Identifying the Goal 554
Creating the Database 556
Establishing the Site 558
Coding the Non-JavaScript Version 559
Creating the Ajax Resources 569
Adding the JavaScript 572
Completing this Example 592
Review and Pursue 593
Wrapping Up 594

Index 595

INTRODUCTION

JavaScript is one of the most widely used programming languages today, found on almost every Web page (certainly all the new ones). Over the past ten years, between economic changes and expansions in how JavaScript is used, more and more Web developers and designers are expected to know this language. These facts make it all the more ironic that so few people respect JavaScript as the true programming language that it is. Furthermore, many books still present JavaScript in a legacy manner, as a technology to be used piecemeal to implement gimmicks and distractions. This book was written to address these problems, presenting JavaScript in a way that you can easily understand, actually master, and appropriately utilize as a productive asset in today's dynamic Web sites.

WHO THIS BOOK IS FOR

This book was written primarily with two types of readers in mind:

- Those who don't know JavaScript at all (and perhaps have never done any programming)
- Those who may have played with JavaScript some, but don't have a solid understanding of why one does what one does in the language.

You may be a Web developer who has written code in other languages but merely dabbled with JavaScript. Or, you may be a Web designer, with a graphical focus but an increasing need to learn JavaScript. Whatever the case, if you have a sincere interest in understanding modern JavaScript and knowing how to use it, well, then this book is for you.

WHAT YOU WILL LEARN

By reading this book, and trying the many examples, you will come to comprehend what JavaScript is and how to reliably program with it, regardless of the task. The book's content is organized in three sections.

PART 1: GETTING STARTED

The first part of the book starts with JavaScript's history and its role in today's Web. You'll also learn the fundamental terms and concepts, particularly when it comes to using JavaScript with HTML in a Web page. The last chapter in Part 1 thoroughly covers the types of tools you'll need to develop, design, debug, and test JavaScript code.

PART 2: JAVASCRIPT FUNDAMENTALS

The bulk of the book is in this second part, which teaches the core components of the language. These fundamentals include the kinds of data you'll work with, operators and control structures, defining your own functions, handling events, and Ajax. Two chapters focus on the browser and HTML forms.

PART 3: NEXT STEPS

All books have their limits, and this book purposefully stops short of trying to cover everything, or attempting to turn you into a true JavaScript "ninja." But in the third part of the book, you will be introduced to what your next logical steps should be in your development as a JavaScript programmer. One chapter is on frameworks, another is on advanced JavaScript concepts, and a third walks through a real-world integration of JavaScript and PHP for a practical Web application.

THE CORRESPONDING WEB SITE

My Web site can be found at `www.LarryUllman.com`. To find the materials specific to this book, click on *Books By Topic* at the top of the page, and then select *JavaScript > Modern JavaScript: Develop and Design*. On the first page that comes up you will find all of the code used in the book. There are also links to errata (errors found) and more information that pertains directly to this book.

The whole site is actually a WordPress blog and you'll find lots of other useful information there, in various categories. The unique tag for this book is *jsdd*, meaning that `www.larryullman.com/tag/jsdd/` will list everything on the site that might be useful and significant to you. While you're at the site, I recommend that you also sign up for my free newsletter, through which I share useful resources, answer questions, and occasionally give away free books.

The book has a corresponding support forum at `www.LarryUllman.com/forums/`. You are encouraged to ask questions there when you need help. You can also follow up on the "Review and Pursue" sections through the forums.

LET'S GET STARTED

With a quick introduction behind you (and kudos for giving it a read), let's get on with the show. In very first chapter, you'll learn quite a bit about JavaScript as a language and the changing role it has had in the history of Web development. There's no programming to be done there, but you'll get a sense of both the big picture and the current landscape, which are important in going forward.

WELCOME TO JAVASCRIPT

A great thing about programming with JavaScript is that most, if not all, of the tools you'll need are completely free. That's particularly reassuring, as you'll want a lot of the following items in order to develop using JavaScript in a productive and reliable way. Chapter 3, Tools of the Trade, goes into the following categories in much more detail.

BROWSERS

Presumably, you already have at least one Web browser, but you'll want several. All the key modern browsers are free and should be used: Chrome, Firefox, Safari, Opera, and even Internet Explorer.

TEXT EDITOR

To write JavaScript code, you can use almost any text editor, although some are clearly better than others. The quick recommendations are Notepad++ on Windows and BBEdit or TextMate on Mac OS X.

IDE

If you prefer an all-in-one tool to a text editor, select an Integrated Development Environment (IDE). The free Aptana Studio is wonderful and runs on most platforms; fine commercial alternatives exist, too.

DEBUGGER

Debugging is a big facet of all programming, and better debugging tools means less stress and a faster development time. Firebug is the clear champion here, although many browsers now have sufficiently good debugging tools built in.

WEB SERVER

Examples in two chapters require a PHP-enabled Web server, plus a MySQL database. If you don't have a live Web site with these already, you can download and install the free XAMPP for Windows or MAMP for Mac OS X.

4

SIMPLE **VARIABLE TYPES**

All programming comes down to taking *some action* with *some data*. In this chapter, the focus is on the data side of the equation, represented by variables. Even if you've never done any programming, you're probably familiar with the concept of a variable: a temporary storage container. This chapter starts with the basics of variables in JavaScript, and then covers number, string, and Boolean variables. Along the way you'll find plenty of real-world code, representing some of the actions you will take with these simple variable types.

BASICS OF VARIABLES

I think it's easiest to grasp variables by starting with so-called "simple" variables, also called "primitive" variable types. By *simple*, I mean variables that only store a single piece of information at a time. For example, a numeric variable stores just a single number; a string, just a sequence of zero or more quoted characters. Simple variables will be the focus in this chapter, with more advanced alternatives—such as arrays and objects—coming in Chapter 6, Complex Variable Types.

To be completely accurate, it's the *values* in JavaScript that are typed, not the variables. Further, many values in JavaScript can be represented as either a *literal* or an *object*. But I don't want to overwhelm you with technical details already, especially if they won't impact your actual programming. Instead, let's focus on this line of code:

```
var myVar = 'easy peasy';
```

TIP: Remember that you can practice much of the JavaScript in this chapter using your browser's console window.

That's a standard and fundamental line of JavaScript programming, declaring a variable named `myVar`, and assigning to it the string *easy peasy*. The next few pages will look at the four components of this one line in detail:

- `var`, used to declare a variable
- the variable's name
- `=`, the assignment operator
- the variable's value

DECLARING VARIABLES

To declare a variable is to formally announce its existence. In many languages, such as C and ActionScript, you must declare a variable prior to referencing it. JavaScript does not *require* you to declare variables, you can just immediately begin referencing them, as in:

```
quantity = 14;
```

(The semicolon is used to terminate a statement. It's not required, but you should always use it.)

Now, to clarify, you don't *have to* declare variables in JavaScript, but *you actually should*. To do that, use the `var` keyword:

```
var fullName;
```

or

```
var fullName = 'Larry Ullman';
```

The distinction between using `var` and not using `var` has to do with the variable's *scope*, a topic that will mean more once you begin defining your own functions (see Chapter 7, Creating Functions). Undeclared variables—those referenced for the first time without using `var`—will have *global* scope by default, and global variables are frowned upon (see the sidebar for more).

Also understand that whether or not you assign a value to the variable when it's declared has no impact on its scope. Both lines above used to declare the `fullName` variable result in a variable with the same scope.

As discussed in Chapter 1, (Re-)Introducing JavaScript, JavaScript is a *weakly typed language*, meaning that variables are not strictly confined to one type or another. Neither of the above uses of `fullName` decree that the variable is a string. With either of those lines of code, this next line will not cause a syntax error:

```
fullName = 2;
```

That line would most likely cause a logical or run-time error, as other code would expect that `fullName` is a string, but the larger point is that a JavaScript variable isn't typed but has a type based upon its value. If `fullName` stores a quoted sequence of zero or more characters, then `fullName` is said to be a string; if `fullName` stores 2, then it's said to be a number.

Note that each variable is only declared once, but you can use `var` to declare multiple variables at the same time:

```
var firstName, lastName;
```

You can even declare multiple variables at the same time while simultaneously assigning values:

```
var firstName = 'Larry', lastName = 'Ullman';
```

GLOBAL **VARIABLES**

All variables have a *scope*, which is the realm in which they exist. As you'll see in Chapter 7, variables declared within a function have *function-level* scope: They only exist within that function. Other languages, but not JavaScript (currently), have *block-level* scope, where a variable can be declared and only exist between a pair of curly braces. Variables declared outside of any function, or referenced without any use of `var`, have *global* scope. There are a few reasons to avoid using global variables.

First, as a general rule of programming, applications should only do the bare minimum of what's required. If a variable does not absolutely need to be global, it shouldn't be. Second, global variables can have an adverse effect on performance, because the application will have to constantly maintain that variable's existence, even when the variable is not being used. By comparison, function variables will only exist during that function's execution (i.e., when the function is called). Third, global variables can cause run-time and logical errors should they conflict with other global variables. This can happen if your code has a variable with the same name as a poorly designed library you might also be including in the same page.

All this being said, understand that for the next few chapters, you will occasionally be using global variables in your code. This is because variables declared outside of any function, even when using the `var` keyword, will also have global scope, and you won't have user-defined functions yet. Still, while it's best not to use global variables, using them is not a terrible, horrible thing, and it's much better to *knowingly* create a global variable than to *accidentally* do so.

You'll rarely see this done in the book, as I will want to better focus on each variable declaration, but lines like that one are common in real-world JavaScript code.

As a final note on the var keyword, you should always declare your variables as soon as possible in your code, within the scope in which they are needed. Variables declared outside of any functions should be declared at the top of the code; variables declared within a function definition should be declared as the first thing within that function's code. The technical reason for this is because of something called "hoisting," but declaring variables as soon as possible is also standard practice in languages without hoisting issues.

VARIABLE NAMES

In order to create a variable, you must give it a name, also called an *identifier*. The rules for names in JavaScript are:

- The name must start with a letter, the underscore, or a dollar sign.
- The rest of the name can contain any combination of letters, underscores, and numbers (along with some other, less common characters).
- You cannot use spaces, punctuation, or any other characters.
- You cannot use a reserved JavaScript word.
- Names are case-sensitive.

This last rule is an important one, and can be a frequent cause of problems. The best way to minimize problems is to use a consistent naming scheme. With an object-oriented language like JavaScript, it's conventional to use "camel-case" syntax, where words within a name are broken up by a capital letter:

- `fullName`
- `streetAddress`
- `monthlyPayment`

In procedural programming languages, the underscore is often used to break up words. In procedural PHP, for example, I would write `$full_name` and `$street_address`. In JavaScript, camel-case is conventional, but the most important criterion is that you choose a style and stick with it.

As a final note, you should not use an existing variable's name for your variable. For example, when JavaScript runs in the browser, the browser will provide some variables, such as `document` and `window`. Both of these are quite important, and you wouldn't want to override them by creating your own variables with those names. You don't need to memorize a list of browser-provided variables, however; just try to be unique and descriptive with your variable names (e.g., `theDocument` and `theWindow` would work fine).

ASSIGNING VALUES

As you probably already know or guessed from what you've seen in this book or online, a single equals sign is the assignment operator, used to assign a value on the right to the variable on the left. Here is the declaration of, and assignment to, a numeric variable:

```
var rate;
rate = 5.25;
```

This can be condensed into a single line:

```
var rate = 5.25;
```

That one line not only declares a variable, but *initializes* it: provides an initial value. You do not have to initialize variables when you declare them, but sometimes it will make sense to.

SIMPLE VALUE TYPES

JavaScript recognizes several "simple" types of values that can be assigned to variables, starting with numbers, strings, and Booleans. A number is exactly what you'd expect: any quantity of digits with or without a single decimal point. Numeric values are never quoted and may contain digits, a single decimal point, a plus or minus, and possibly the letter "e" (for exponential notation). Numeric values do not contain commas, as would be used to indicate thousands.

A string is any sequence of zero or more quoted characters. You can use single or double quotation marks, but you must use the same type to end the string as you used to begin it:

- 'This is a string.'
- "This is also a string."

If you need to include a single or double quotation mark within the string, you can either use the other mark type to delineate the string or *escape* the potentially problematic character by prefacing it with a backslash:

- "I've got an idea."
- 'Chapter 4, "Simple Variable Types"'

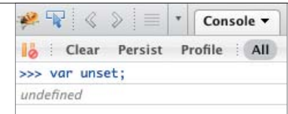


FIGURE 4.1 Because this variable has not yet been assigned a value, its value is `undefined`.

- 'I\'ve got an idea.'
- "Chapter 4, \"Simple Variable Types\""

What will not work is:

- 'I've got an idea.'
- "Chapter 4, "Simple Variable Types""

Note that a string does not need to have any characters in it: Both '' and "" are valid strings, called *empty* strings.

JavaScript also has Boolean values: `true` and `false`. As JavaScript is a case-sensitive language, you must use *true* and *false*, not *True* or *TRUE* or *False* or *FALSE*.

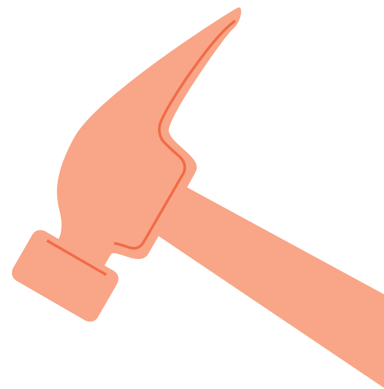
Two more simple, yet special, values are `null` and `undefined`. Again, these are case-sensitive words. The difference between them is subtle. `null` is a defined non-value and is best used to represent the consequence of an action that has no result. For example, the result of a working Ajax call could be `null`, which is to say that no data was returned.

Conversely, `undefined` is no set value, which is normally the result of inaction. For example, when a variable is declared without being assigned a value, its value will be `undefined` (**Figure 4.1**):

```
var unset; // Currently undefined.
```

Similarly, if a function does not actively return a value, then the returned value is `undefined` (you'll see this in Chapter 7).

Both `null` and `undefined` are not only different from each other, but different from `false`, which is a known and established negative value. As you'll see in Chapter 5, Using Control Structures, when used as the basis of a condition, both `null` and `undefined` are treated as FALSE, as are the number 0 and the empty string. Still, there are differences among them.



TIP: As a reminder, the combination of two slashes together (//) creates a comment in JavaScript.

WORKING WITH NUMBERS

Unlike a lot of languages, JavaScript only has a single number type, used to represent any numerical value, from integers to doubles (i.e., decimals or real numbers) to exponent notation. You can rest assured in knowing that numbers in JavaScript can safely represent values up to around 9 quadrillion!

Let's look at everything you need to know about numbers in JavaScript, from the arithmetic operators to formatting numbers, to using the `Math` object for more sophisticated purposes.

ARITHMETIC OPERATORS

You've already been introduced to one operator: a single equals sign, which is the assignment operator. JavaScript supports the standard arithmetic operators, too (**Table 4.1**).

TABLE 4.1 Arithmetic Operators

SYMBOL	MEANING
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

The modulus operator, in case you're not familiar with it, returns the remainder of a division. For example:

```
var remainder = 7 % 2; // 1;
```

One has to be careful when applying the modulus operator to negative numbers, as the remainder itself will also be negative:

```
var remainder = -7 % 2; // -1
```

These arithmetic operators can be combined with the assignment operator to both perform a calculation and assign the result in one step:

```
var cost = 50; // Dollars
cost *= 0.7373; // Converted to euros
```

You'll frequently come across the increment and decrement operators: ++ and --. The increment operator adds one to the value of the variable; the decrement operator subtracts one:

```
var num = 1;
num++; // 2
num--; // 1
```

These two operators can be used in both *prefix* and *postfix* manners (i.e., before the variable or after it):

```
var num = 1;
num++; // num now equals 2.
++num; // num is now 3.
--num; // num is now 2.
```

A difference between the postfix and prefix versions is a matter of *operator precedence*. The rules of operator precedence dictate the order operations are executed in a multi-operation line. For example, basic math teaches that multiplication and division have a higher precedence than addition and subtraction. Thus:

```
var num = 3 * 2 + 1; // 7, not 9
```

Table 4.2 lists the order of precedence in JavaScript, from highest to lowest, including some operators not yet introduced (I've also omitted a couple of operators that won't be discussed in this book). There's also an issue of *associativity* that I've omitted, as that would be just one more thing you'd have to memorize. In fact, instead of trying to memorize that table, I recommend you use parentheses to force, or just clarify, precedence, without relying upon mastery of these rules. For example:

```
var num = (3 * 2) + 1; // Still 7.
```

That syntax, while two characters longer than the earlier version, has the same net effect but is easier to read and undeniably clear in intent.

Some of the operators in Table 4.2 are *unary*, meaning they apply to only one operand (such as ++ and --); others are binary, applying to two operands (such as addition). In Chapter 5, you'll learn how to use the one *trinary* operator, which has three operands.

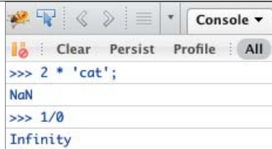


FIGURE 4.2 The result of invalid mathematical operations will be the special values `NaN` and `Infinity`.

TABLE 4.2 Operator Precedence

PRECEDENCE	OPERATOR	NOTE
1	. []	member operators
1	new	creates new objects
2	()	function call
3	++ --	increment and decrement
4	!	logical not
4	+ -	unary positive and negative
4	typeof void delete	
5	* / %	multiplication, division, and modulus
6	+ -	addition and subtraction
8	< <= > >=	comparison
9	== != === !==	equality
13	&&	logical and
14	||	logical or
15	?:	conditional operator
16	= += -= *= /= %= <<= >>= >>>= &= ^= |=	assignment operators

The last thing to know about performing arithmetic in JavaScript is if the result of the arithmetic is invalid, JavaScript will return one of two special values:

- `NaN`, short for *Not a Number*
- `Infinity`

For example, you'll get these results if you attempt to perform arithmetic using strings or when you divide a number by zero, which surprisingly doesn't create an error (**Figure 4.2**). In Chapter 5, you'll learn how to use the `isNaN()` and `isFinite()` functions to verify that values are numbers safe to use as such.

CREATING CALCULATORS

At this point in time, you have enough knowledge to begin using JavaScript to perform real-world mathematical calculations, such as the kinds of things you'd put on a Web site:

- Mortgage and similar loan calculators
- Temperature and other unit conversions
- Interest or investment calculators

For this particular example, let's create an e-commerce tool that will calculate the total of an order, including tax, and minus any discount (**Figure** 4.3). The most relevant HTML is:

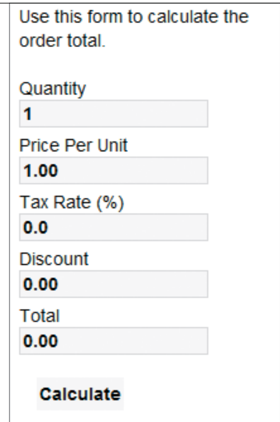


FIGURE 4.3 A simple calculator.

```
<div><label for="quantity">Quantity</label><input type="number"
→ name="quantity" id="quantity" value="1" min="1" required></div>
<div><label for="price">Price Per Unit</label><input type="text"
→ name="price" id="price" value="1.00" required></div>
<div><label for="tax">Tax Rate (%)</label><input type="text"
→ name="tax" id="tax" value="0.0" required></div>
<div><label for="discount">Discount</label><input type="text"
→ name="discount" id="discount" value="0.00" required></div>
<div><label for="total">Total</label><input type="text" name="total"
→ id="total" value="0.00"></div>
<div><input type="submit" value="Calculate" id="submit"></div>
```

That would go in a page named `shopping.html`, which includes the `shopping.js` JavaScript file, to be written in subsequent steps. You'll notice that the HTML form makes use of the HTML5 number input type for the quantity, with a minimum value. The other types are simply text, as the number type doesn't deal well with decimals. Each input is given a default value, and set as required. Remember that as Chapter 2, JavaScript in Action, explains, browsers that don't support HTML5 will treat unknown types as text elements and ignore the unknown properties. The final text element will be updated with the results of the calculation.

To create a calculator:

1. Create a new JavaScript file in your text editor or IDE, to be named `shopping.js`.

2. Begin defining the `calculate()` function:

```
function calculate() {
    'use strict';
```

   This function will be called when the user clicks the submit button. It does the actual work.

3. Declare a variable for storing the order total:

```
var total;
```

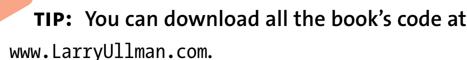
   As mentioned previously, you should generally declare variables as soon as you can, such as the first line of a function definition. Here, a variable named `total` is declared but not initialized.

4. Get references to the form values:

```
var quantity = document.getElementById('quantity').value;
var price = document.getElementById('price').value;
var tax = document.getElementById('tax').value;
var discount = document.getElementById('discount').value;
```

   In these four lines of code, the values of the various form elements are assigned to local variables. Note that in the Chapter 2 example, variables were assigned references to the form elements, and then the element values were later checked. Here, the value is directly assigned to the variable.

   At this point in time, one would also perform validation of these values, prior to doing any calculations. But as Chapter 5 more formally covers the knowledge needed to perform validation, I'm skipping this otherwise needed step in this example.



TIP: You can download all the book's code at `www.LarryUllman.com`.

5. Calculate the initial total:

```
total = quantity * price;
```

The `total` variable is first assigned the value of the quantity times the price, using the multiplication operator.

6. Factor in the tax rate:

```
tax /= 100;

tax++;

total *= tax;
```

There are a couple of ways one can calculate and add in the tax. The first, shown here, is to change the tax rate from a percent (say 5.25%) to a decimal (0.0525). Next, add one to the decimal (1.0525). Finally, multiply this number times the total. You'll see that the division-assignment, incrementation, and multiplication-assignment operators are used here as shorthand. This code could also be written more formally:

```
tax = tax/100;

tax = tax + 1;

total = total * tax;
```

You could also make use of precedence and parentheses to perform all these calculations in one line.

An alternative way to calculate the tax would be to convert it to decimal, multiply that value times the total, and then add that result to the total.

7. Factor in the discount:

```
total -= discount;
```

The discount is just being subtracted from the total.

8. Display the total in the form:

```
document.getElementById('total').value = total;
```

The `value` attribute can also be used to *assign* a value to a text form input. Using this approach, you can easily reflect data back to the user. In later chapters, you'll learn how to display information on the HTML page using DOM manipulation, rather than setting the values of form inputs.

9. Return `false` to prevent submission of the form:

```
return false;
```

The function must return a value of `false` to prevent the form from actually being submitted (to the page named by the form's `action` attribute).

10. Complete the function:

```
} // End of calculate() function.
```

11. Define the `init()` function:

```
function init() {
    'use strict';
    var theForm = document.getElementById('theForm');
    theForm.onsubmit = calculate;
} // End of init() function.
```

The `init()` function will be called when the window triggers a load event (see Step 12). The function needs to add an event listener to the form's submission, so that when the form is submitted, the `calculate()` function will be called. To do that, the function gets a reference to the form, by calling the `document` object's `getElementById()` method, providing it with the unique ID value of the form. Then the variable's `onsubmit` property is assigned the value *calculate*, as explained in Chapter 2.

12. Add an event listener to the window's load event:

```
window.onload = init;
```

This code was also explained in Chapter 2. It says that when the window has loaded, the `init()` function should be called.

Use this form to calculate the order total.

Quantity
12
Price Per Unit
1.99
Tax Rate (%)
5.25
Discount
3.00
Total
22.133699999999997

Calculate

Use this form to calculate the order total.

Quantity
cat
Price Per Unit
1.99
Tax Rate (%)
5.25
Discount
3.00
Total
NaN

Calculate

FIGURE 4.4 The result of the total order calculation.

FIGURE 4.5 Performing arithmetic with invalid values, such as a quantity of *cat*, will result in a total of `NaN`.

It's a minor point, as you can organize your scripts in rather flexible ways, but this line is last as it references the `init()` function, defined in Step 12, so that definition should theoretically come before this line. That function references `calculate()`, so the `calculate()` function's definition is placed before the `init()` function definition. You don't have to organize your code this way, but I prefer to.

13. Save the file as `shopping.js`, in a `js` directory next to `shopping.html`, and test in your Web browser (**Figure 4.4**).

    Play with the numbers, including invalid values (**Figure 4.5**), and retest the calculator until you're comfortable with how arithmetic works in JavaScript.

FORMATTING NUMBERS

Although the previous example is perfectly useful, and certainly a good start, there are several ways in which it can be improved. For example, as written, no checks are made to ensure that the user enters values in all the form elements, let alone that those values are numeric (Figure 4.5) or, more precisely, positive numbers. That knowledge will be taught in the next chapter, which discusses conditionals, comparison operators, and so forth. Another problem, which can be addressed here, is that you can't expect someone to pay, say, 22.1336999 (Figure 4.4). To improve the professionalism of the calculator, formatting the calculated total to two decimal points would be best.

A number in JavaScript is not just a number, but is also an object of type `Number`. As an object, a number has built-in methods, such as `toFixed()`. This method returns a number with a set number of digits to the right of a decimal point:

```
var num = 4095.3892;
num.toFixed(3); // 4095.389
```

Note that this method only returns the formatted number; it does not change the original value. To do that, you'd need to assign the result back to the variable, thereby replacing its original value:

```
num = num.toFixed(3);
```

If you don't provide an argument to the `toFixed()` method, it defaults to 0:

```
var num = 4095.3892;
num.toFixed(3); // 4095
```

The method can round up to 20 digits.

Similar to `toFixed()` is `toPrecision()`. It takes an argument dictating the total number of significant digits, which may or may not include those after the decimal.

Let's apply this information to the calculator in order to add some better formatting to the total.

To format a number:

1. Open `shopping.js` in your text editor or IDE, if it is not already.
2. After factoring in the discount, but before showing the total amount, format the total to two decimals:

   ```
   total = total.toFixed(2);
   ```

   This one line will take care of formatting the decimal places. Remember that the returned result must be assigned back to the variable in order for it to be represented upon later uses.

   Alternatively, you could just call `total.toFixed(2)` when assigning the value to the total form element.

FIGURE 4.6 The same input as in Figure 4.4 now generates a more appropriate result.

FIGURE 4.7 The area of a circle, πr2, is calculated using the Math.PI constant.

3. Save the file, reload the HTML page, and test it in your Web browser (**Figure 4.6**).

An even better way of formatting the number would be to add commands indicating thousands, but that requires more logic than can be understood at this point in the book.

THE MATH OBJECT

You just saw that numbers in JavaScript can also be treated as objects of type Number, with a couple of built-in methods that can be used to manipulate them. Another way to manipulate numbers in JavaScript involves the Math object. Unlike Number, you do not create a variable of type Math, but use the Math object directly. The Math object is a global object in JavaScript, meaning it's always available for you to use.

The Math object has several predefined *constants*, such as π, which is 3.14... and E, which is 2.71... A constant, unlike a variable, has a fixed value. Conventionally, constants are written in all uppercase letters, as shown. Referencing an object's constant uses the same dot syntax as you would to reference one of its methods: Math.PI, Math.E, and so forth. Therefore, to calculate the area of a circle, you could use (**Figure 4.7**):

```
var radius = 20;
var area = Math.PI * radius * radius;
```

Use this form to calculate the volume of a sphere.

Radius

12

Volume

7238.2295

Calculate

FIGURE 4.8 This calculator determines and displays the volume of a sphere given a specific radius.

The `Math` object also has several predefined methods, just a few of which are:

- `abs()`, which returns the absolute value of a number
- `ceil()`, which rounds up to the nearest integer
- `floor()`, which rounds down to the nearest integer
- `max()`, which returns the largest of zero or more numbers
- `min()`, which returns the smallest of zero or more numbers
- `pow()`, which returns one number to the power of another number
- `round()`, which returns a number rounded to the nearest integer
- `random()`, which returns a pseudo-random number between 0 (inclusive) and 1 (exclusive)

There are also several trigonometric methods like `sin()` and `cos()`.

Another way of writing the formula for determining the area of a circle is:

```
var radius = 20;
var area = Math.PI * Math.pow(radius, 2);
```

To apply this new information, let's create a new calculator that calculates the volume of a sphere, based upon a user-entered radius. That formula is:

$$volume = 4/3 * \pi * radius^3$$

Besides using the π constant and the `pow()` method, this next bit of JavaScript will also apply the `abs()` method to ensure that only a positive radius is used for the calculation (**Figure 4.8**). The relevant HTML is:

```
<div><label for="radius">Radius</label><input type="text"
→ name="radius" id="radius" required></div>
<div><label for="volume">Volume</label><input type="text"
→ name="volume" id="volume"></div>
<div><input type="submit" value="Calculate" id="submit"></div>
```

The HTML page includes the `sphere.js` JavaScript file, to be written in subsequent steps.

To calculate the volume of a sphere:

1. Create a new JavaScript file in your text editor or IDE, to be named `sphere.js`.

2. Begin defining the `calculate()` function:

   ```
   function calculate() {
       'use strict';
       var volume;
   ```

   Within the function, a variable named `volume` is declared, but not initialized.

3. Get a reference to the form's radius value:

   ```
   var radius = document.getElementById('radius').value;
   ```

   Again, this code closely replicates that in `shopping.js`, although there's only one form value to retrieve.

4. Make sure that the radius is a positive number:

   ```
   radius = Math.abs(radius);
   ```

   Applying the `abs()` method of the `Math` object to a number guarantees a positive number without having to use a conditional to test for that.

5. Calculate the volume:

   ```
   volume = (4/3) * Math.PI * Math.pow(radius, 3);
   ```

   The volume of a sphere is four-thirds times π times the radius to the third power. This one line performs that entire calculation, using the `Math` object twice. The division of four by three is wrapped in parentheses to clarify the formula, although in this case the result would be the same without the parentheses.

6. Format the volume to four decimals:

   ```
   volume = volume.toFixed(4);
   ```

   Remember that the `toFixed()` method is part of `Number`, which means it's called from the `volume` variable, not from the `Math` object.

7. Display the volume:

```
document.getElementById('volume').value = volume;
```

This code is the same as in the previous example, but obviously referencing a different form element.

8. Return `false` to prevent the form's submission, and complete the function:

```
    return false;
} // End of calculate() function.
```

9. Add an event listener to the form:

```
function init() {
    'use strict';
    document.getElementById('calcForm').onsubmit = calculate;
} // End of init() function.
window.onload = init;
```

This is the same code used in `shopping.js`. As in that example, when the form is submitted, the `calculate()` function will be called.

10. Save the file as `sphere.js`, in a `js` directory next to `sphere.html`, and test it in your Web browser.

WORKING WITH STRINGS

Strings and numbers are two of the most common types used in JavaScript, and both are easy to comprehend and use. You've seen the fundamentals when it comes to numbers—and there's not all that much to it, really, so now it's time to look at strings in more detail.

CREATING STRINGS

Informally, you've already witnessed how strings are created: just quote anything. As with a number, once you have a string value, you also have predefined methods that can be used to manipulate that value. Unlike numbers, though, strings have a

lot more methods, and even a property you'll commonly use: `length`. The `length` property stores the number of characters found in the string, including empty spaces:

```
var fullName = 'Larry Ullman';
fullName.length; // 12
```

If you're following this book sequentially, you'll have already seen this in Chapter 2:

```
var email = document.getElementById('email');
if ( (email.value.length > 0) { ...
```

What you're actually seeing here is the beauty of object-oriented programming: A string is a string, with all the functionality that comes with it, regardless of how the string was created. The assignment to the `email` variable starts with the `document` object, which is a representation of the page's HTML. That object has a `getElementById()` method, which returns an HTML element. The specific element returned by that line is a text input, in other words, a text object. This is assigned to `email`. That object has a `value` property for finding the text input's value (or for setting its value). Since the value returned by that property is a string, you can then refer to its `length` property. Thanks to the ability to chain object notation, this could be reduced to one line:

```
if ( (document.getElementById('email').value.length > 0) { ...
```

DECONSTRUCTING STRINGS

Once you've created a string, you can deconstruct it—break it into pieces—in a number of ways. As a string is just a sequence of `length` characters, you can reference individual characters using the `charAt()` method. This method takes an *index* as its first argument, an index being the position of the character in the string. The trick to using indexes is that they begin at 0, not 1 (this is common to indexes of all types across all programming languages). Thus, the first character of string `fullName` can be retrieved using `fullName.charAt(0)`. And a string's last character will be indexed at `length - 1`:

```
var fullName = 'Larry Ullman';
fullName.charAt(0); // L
fullName.charAt(11); // n
```

Sometimes you don't want to know what character is at a specific location in the string, but rather if a character is found in the string at all. For this need, use the `indexOf()` method. This method returns the indexed position where the character is first found:

```
var fullName = 'Larry Ullman';
fullName.indexOf('L'); // 0
fullName.indexOf('a'); // 1
fullName.indexOf(' '); // 5
```

The first argument can be more than a single character, letting you see if entire words are found within the string. In that case, the method returns the indexed position where the word begins in the string:

```
var language = 'JavaScript';
language.indexOf('Script'); // 4
```

The `indexOf()` method takes an optional second argument, which is a location to begin searching in the string. By default, this is `0`:

```
var language = 'JavaScript';
language.indexOf('a'); // 1
language.indexOf('a', 2); // 3
```

However you use `indexOf()`, if the character or characters—the *needle*—is not found within the string (the *haystack*), the method returns `-1`. Also, `indexOf()` performs a case-sensitive search:

```
var language = 'JavaScript';
language.indexOf('script'); // -1
```

Another way to look for needles within a string haystack is to use `lastIndexOf()`, which goes backward through the string. Its second argument is also optional, and indicates the starting point, but the search again goes backward from that starting point, not forward:

```
var fullName = 'Larry Ullman';
fullName.indexOf('a'); // 1
```

```
fullName.lastIndexOf('a'); // 10
fullName.lastIndexOf('a', 5); // 1
```

To pull a substring out of a string, there's the `slice()` method. Its first argument is the index position to begin at. Its optional second argument is the indexed position where to stop. Without this second argument, the substring will continue until the end of the string:

```
var language = 'JavaScript';
language.slice(4); // Script
language.slice(0,4); // Java
```

A nice trick with `slice()` is that you can provide a negative second argument, which indicates the index at which to stop, counting backward from the end of the string. If you provide a negative starting point, the slice will begin at that indexed position, counting backward from the end of the string:

```
var language = 'JavaScript';
language.slice(0,-6); // Java
language.slice(-6); // Script
```

However you use `slice()`, this method only returns a new string, without affecting the value of the original.

JavaScript also has a `substring()` method, which uses the same arguments as `slice()`, but it has some unexpected behaviors, and it's recommended that you use `slice()` instead.

JavaScript has another string method for retrieving substrings: the aptly named `substr()`. Its first argument is the starting index for the substring, but the second is the number of characters to be included in the substring, not the terminating index. In theory, you can provide negative values for each, thereby changing both the starting and ending positions to be relative to the end of the string, but Internet Explorer doesn't accept negative starting positions.

NOTE: **In Chapter 6, you'll learn about the** `split()` **method, which breaks a string into an array of strings.**

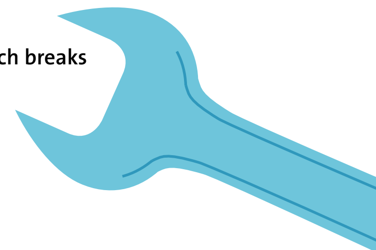


FIGURE 4.9 The HTML form, as it works in Internet Explorer.

To test using `slice()`, let's create some JavaScript code that limits the amount of data that can be submitted by a textarea. For the time being, a second textarea will show the restricted string; in Chapter 8, Event Handling, you'll learn how to dynamically restrict the amount of text entered in a text area in real time. The relevant HTML for this example is:

```
<div><label for="comments">Comments</label><textarea name="comments"
→ id="comments" maxlength="100" required></textarea></div>
<div><label for="count">Character Count</label><input type="number"
→ name="count" id="count"></div>
<div><label for="result">Result</label><textarea name="result"
→ id="result"></textarea></div>
<div><input type="submit" value="Submit" id="submit"></div>
```

The HTML form has one textarea for the user's input, a text input indicating the number of characters used, and another textarea showing the truncated result. To make the truncated text more professional, it'll be broken on the final space before the character limit (**Figure 4.9**), rather than having the text broken midword. The page, named `text.html`, includes the `text.js` JavaScript file, to be written in subsequent steps.

To deconstruct strings:

1. Create a new JavaScript file in your text editor or IDE, to be named `text.js`.

2. Begin defining the `limitText()` function:

   ```
   function limitText() {
       'use strict';
       var limitedText;
   ```

   The `limitedText` variable will be used to store the edited version of the user-supplied text.

3. Retrieve the original text:

   ```
   var originalText = document.getElementById('comments').value;
   ```

   The original text comes from the first textarea in the form and is assigned to `originalText` here.

4. Find the last space before the one-hundredth character in the original text:

```
var lastSpace = originalText.lastIndexOf(' ', 100);
```

To find the last occurrence of a character in a string, use the `lastIndexOf()` method, applied to the original string. This script is not looking for the absolute last space, though, just the final space before the hundredth character, so 100 is provided as the second argument to `lastIndexOf()`, meaning that the search will begin at the index of 100 and work backward.

5. Trim the text to that spot:

```
limitedText = originalText.slice(0, lastSpace);
```

Next, a substring from `originalText` is assigned to `limitedText`, starting at the beginning of the string—index of 0—and stopping at the previously found space.

6. Show the user the number of characters submitted:

```
document.getElementById('count').value = originalText.length;
```

To indicate that the user submitted too much data, the original character count will be shown in a text input.

7. Display the limited text:

```
document.getElementById('result').value = limitedText;
```

The value of the second textarea is updated with the edited string.

8. Return `false` and complete the function:

```
    return false;
} // End of limitText() function.
```

TIP: It'd be more professional to break the text on a space or comma or the end of a sentence, but that capability is beyond this point in the book.

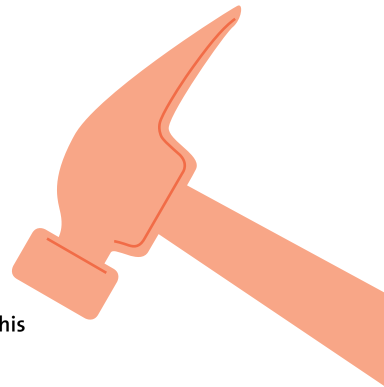


FIGURE 4.10 In Chrome, which supports the textarea's `maxlength` attribute, only 100 characters can be submitted, but the partial word is still chopped off.

9. Add an event listener to the form:

```
function init() {
    'use strict';
    document.getElementById('calcForm').onsubmit = limitText;
} // End of init() function.
window.onload = init;
```

   This is the same basic code used in the previous example. When the form is submitted, the `limitText()` function will be called.

10. Save the file as `text.js`, in a `js` directory next to `text.html`, and test it in your Web browser (Figure 4.9).

   Try using different strings (**Figure 4.10**), and retest, to make sure it's working as it should.

MANIPULATING STRINGS

The most common way to manipulate a string is to change its value using *concatenation*. Concatenation is like addition for strings, adding more characters onto existing ones. In fact, the concatenation operator in JavaScript is also the arithmetic addition operator:

```
var message = 'Hello';
message = message + ', World! ';
```

As with the arithmetic addition, you can combine the plus sign with the assignment operator (=) into a single step:

```
var message = 'Hello';
message += ', World! ';
```

This functionality is duplicated by the `concat()` method, although it's less commonly used. This method takes one or more strings to be appended to the string:

```
var address = '100 Main Street';
address.concat(' Anytown', ' ST', ' 12345', ' US');
```

CONSTANTS

Many programming languages have the concept of a *constant*: a single value that cannot be changed (depending upon how and where the constant was created, and depending upon the language, the constant can have other qualities, too). In theory, JavaScript has the ability to create a constant, using this code:

```
const NAME = value;
```

The same naming rules as those for variables apply to constants, but constants are conventionally written in all uppercase letters, using underscores to separate words. Regardless, the `const` keyword is not supported across all browsers; specifically, Internet Explorer doesn't recognize it. There are ways to fake a constant, but that requires code well beyond what you would know at this point. The end result is that you shouldn't plan on creating your own constants in JavaScript code.

On the other hand, many built-in JavaScript objects have their defined constants, like the `Number` object's `MAX_VALUE`. This constant represents the maximum value that a number can have in the given environment. You'd refer to it using `Number.MAX_VALUE`.

Two methods exist to simply change the case of the string's characters: `toLowerCase()` and `toUpperCase()`. You can apply these to a string prior to using one of the previously mentioned methods, in order to fake case-insensitive searches:

```
var language = 'JavaScript';
language.indexOf('script'); // -1, aka not found
language.toLowerCase().indexOf('script'); // 4
```

Added to JavaScript in version 1.8.1 is the `trim()` method, which removes extra spaces from both ends of a string. It's supported in more current browsers—Chrome, Firefox 3.5 and up, IE9 and above, Safari 5 and up, and Opera 10.5 and above, but isn't available on older ones.

Note that, as with `slice()` and the other methods already covered, `toLowerCase()`, `toUpperCase()`, and `trim()` do not affect the original string, they only return a modified version of that string. Concatenation, however, does alter the original.

First Name
Larry
Last Name
Ullman
Formatted Name
Ullman, Larry
Submit

FIGURE 4.11 The values entered in the first two inputs are concatenated together to create a formatted name.

To test this new information, this next example will take a person's first and last names, and then format them as *Surname, First Name* (**Figure 4.11**). The relevant HTML is:

```
<div><label for="firstName">First Name</label><input type="text"
→ name="firstName" id="firstName" required></div>
<div><label for="lastName">Last Name</label><input type="text"
→ name="lastName" id="lastName" required></div>
<div><label for="result">Formatted Name</label><input type="text"
→ name="result" id="result" required></div>
<div><input type="submit" value="Submit" id="submit"></div>
```

This would go into an HTML page named `names.html`, which includes the `names.js` JavaScript file, to be written in subsequent steps. By this point in the chapter, this should be a simple and obvious exercise for you.

To manipulate strings:

1. Create a new JavaScript file in your text editor or IDE, to be named `names.js`.

2. Begin defining the `formatNames()` function:

   ```
   function formatNames() {
       'use strict';
       var formattedName;
   ```

   The `formattedName` variable will be used to store the formatted version of the user's name.

3. Retrieve the user's first and last names:

   ```
   var firstName = document.getElementById('firstName').value;
   var lastName = document.getElementById('lastName').value;
   ```

4. Create the formatted name:

   ```
   formattedName = lastName + ', ' + firstName;
   ```

   To create the formatted name, assign to the `formattedName` variable the `lastName` plus a comma plus a space, plus the `firstName`. There are other ways of performing this manipulation, such as:

```
formattedName = lastName;
formattedName += ', ';
formattedName += firstName;
```

That code would probably perform worse, though, than the one-line option.

5. Display the formatted name:

```
document.getElementById('result').value = formattedName;
```

6. Return `false` and complete the function:

```
    return false;
} // End of formatNames() function.
```

7. Add an event listener to the form:

```
function init() {
    'use strict';
    document.getElementById('calcForm').onsubmit = formatNames;
} // End of init() function.
window.onload = init;
```

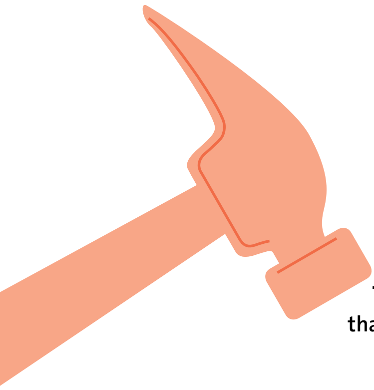
When the form is submitted, the `formatNames()` function will be called.

8. Save the file as `names.js`, in a `js` directory next to `names.html`, and test it in your Web browser (Figure 4.11).

ESCAPE SEQUENCES

Another thing to understand about strings in JavaScript is that they have certain meaningful *escape sequences*. You've already seen two examples of this: to use a type of quotation mark (single or double) within a string delimited by that same type, the inserted quotation mark must be prefaced with a backslash:

- 'I\'ve got an idea.'
- "Chapter 4, \"Simple Variable Types\""



Three other meaningful escape sequences are:

- `\n`, a new line
- `\r`, a carriage return
- `\\`, a literal backslash

Note that these work within either single or double quotation marks (unlike, for example, in PHP, where they only apply within double quotation marks).

TIP: When a user presses Enter or Return within a textarea, that translates to `\n` in a corresponding JavaScript string.

PERFORMING TYPE CONVERSIONS

Because JavaScript is weakly typed, different value types can be used together without causing formal errors. In, say, ActionScript, the following would cause an error:

```
var cost:int = 2;
cost += ' dollars';
```

But in JavaScript, you can do that without the browser complaining. That being said, although you *can* use different types together without causing formal errors, it's quite possible to end up with logical errors, which is to say *bugs*, if you're not careful. One complication stems from the fact that the addition operator in math is the same as the concatenation operator for strings. When you add a string to a number, or add a number to a string, JavaScript will convert the number to a string and then concatenate the two. For example, say the shopping example added a shipping value to the total:

```
var shipping = document.getElementById('shipping').value;
total = quantity * price;
tax /= 100;
tax++;
total *= tax;
total += shipping;
```

FIGURE 4.12 Adding the string '5.00' to the total has the impact of concatenation, converting the `total` number into an unusable string.

FIGURE 4.13 How the `parseInt()` function extracts numbers from strings.

By the time JavaScript gets to the final line, `total` is a number, but `shipping` is a string, because it comes from a form's text input. That final line won't have the effect of mathematically adding the shipping to the total but rather concatenating the shipping onto the total (**Figure 4.12**).

This issue doesn't apply to other operators, though. For example, subtraction converts a string to a number and then performs the math, as the shopping example already demonstrated.

To perform math using strings, without worrying about creating bugs, you can forcibly convert the string to a number. There are many ways of doing so, starting with `parseFloat()` and `parseInt()`. These are "top-level" functions, which is to say they are not associated with any object and can be called directly. The first function always returns a floating-point number (aka, a decimal), and the latter, an integer. Both functions take the value to be converted as its first argument. The `parseInt()` function takes the *radix* as the second. The radix is the number's base, as in base-8 (aka, octal), base-10 (decimal), and base-16 (hexadecimal). Although the second argument is optional, you should always provide it to be safe, and will normally use a value of 10:

```
total += parseFloat(shipping, 10);
```

To best use these functions, you should have an understanding of how they work. Both functions begin at the start of the string and extract a number until an invalid numeric character is encountered. If no valid number can be pulled from the start of the value, both functions return `NaN` (**Figure 4.13**):

```
parseInt('20', 10);
parseInt('20.0', 10);
parseInt('20 ducklings', 10);
parseInt('I saw 20 ducklings.', 10);
```

OBJECTS VS. LITERALS

A point that this chapter has thus far ignored is that values can be represented in two ways: as *objects* or as *literals*. All of the examples in this chapter are literals, such as these:

- 2
- 'JavaScript'
- false

This is the most common way for creating simple variable types, but you can create numbers, strings, and Booleans as formal objects, too:

```
var number = new Number(2);

var fullName = new String('JavaScript');

var flag = new Boolean(false);
```

In that code, the corresponding global function—`String`, `Number`, and `Boolean`—is used to create and return an object of the given type.

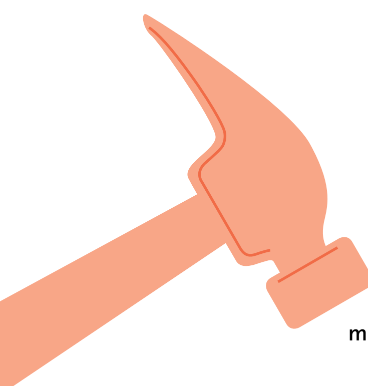
Besides being more complicated to write, creating simple types as objects will actually have slightly worse performance and have some unexpected behaviors. And you can continue to use literals as if they were objects, as many of the examples in this chapter have shown, without formally creating the object. In such cases, when needed, JavaScript will convert the literal value to a corresponding object, call the object's method, and then remove the temporary object.

A trickier way to convert a string to a number is to prepend it with a +:

```
total += +shipping;
```

or

```
total += +(shipping);
```



TIP: You can also convert a string to a number by multiplying it by 1.

Using this unary operator is the fastest solution, in terms of how quickly JavaScript performs the conversion, but is not as clear in terms of programmer readability as `parseInt()` and `parseFloat()`.

Converting from a number to a string is far less likely to cause problems, but you can do so by invoking the `toString()` method:

```
var message = 'Your total is $' + total.toString();
```

The `toString()` method is supported by most objects and returns a string representation of the object itself.

Earlier in the chapter, I mentioned two other meaningful values in JavaScript: `undefined` and `null`. As a gotcha, you should be aware of what happens when an `undefined` or `null` value is used as if it were a number. The `undefined` value translates to `NaN` when used as a number. When a `null` value is used as a number, the result is better, although not great: `null` values are treated as 0 as numbers (**Figure 4.14**). In the next chapter, you'll learn how to verify that a value is numeric prior to attempting to use it as such.

FIGURE 4.14 How arithmetic is handled if `undefined` or `null` is involved.

REVIEW AND PURSUE

Beginning in Part 2: JavaScript Fundamentals, each chapter of this book ends with a "Review and Pursue" section. In these sections, you'll find questions regarding the material just covered and prompts for ways to expand your knowledge and experience on your own. If you have any problems with these sections, either in answering the questions or pursuing your own endeavors, turn to the book's supporting forum (`www.LarryUllman.com/forums/`).

REVIEW

- How do you *declare* a variable?
- What is variable *scope*?
- What are the rules for a variable's name?
- What is the assignment operator?

- What simple types were introduced in this chapter?
- How can you use a single quotation mark within a string? A double quotation mark?
- What does the *= operator do? How about +=? (There are two answers to this last question.) And what about ++?
- What operator can cause bugs when used with a string and a number together?
- What does the `toFixed()` method do?
- What are some of the differences between `Number` objects and the `Math` object?
- What is an empty string?
- What does the `charAt()` method do? What does `indexOf()` do? How about `lastIndexOf()`? What are the arguments to the `indexOf()` and `lastIndexOf()` methods? What happens when you use negative numbers for the second argument to either method?
- What function should you use to pull a substring out of a string and how do you use it?
- What are the various ways you can perform concatenation with strings?
- What are escape sequences?
- What are some of the ways you can convert a string to a number?

PURSUE

- Use a development tool such as Firebug to practice creating and manipulating variables.
- Look up some of JavaScript's reserved words, if you have not already.
- If you're curious, find out what "hoisting" is.
- Create another calculator, such as one that calculates the area of a shape (rectangle, triangle, circle, etc.).

- Look online (e.g., at `https://developer.mozilla.org`) to research all the `Number` and `Math` object properties and methods.
- Look online to learn more about the `String` object and its methods.
- Create another string manipulation example.
- Update the shopping example to add a shipping cost option, and then rework the JavaScript to properly add the shipping amount to the total.
- Test all of this chapter's code in as many browsers and devices as you can to see the various results.

WRAPPING **UP**

In this chapter, you started learning the fundamental lessons of real programming in JavaScript, centered around the simple variable types. Those types include numbers, strings, and Booleans. You learned how to declare variables, how to properly name them, and how to assign them simple values.

Next, the chapter looked into the number type in detail, which starts with basic arithmetic. From there, you saw how to use the `Number` and `Math` object methods in this object-oriented language to perform such commonplace tasks as formatting numbers and rounding them.

After numbers, similar treatment was given to strings: what they are and how to create them. You also learned that there are several methods defined within the `String` object that are usable on any string you have. One of the most common manipulations of strings is *concatenation*, accomplished via the plus sign. Attention was also given to using the backslash as an escaping character.

The chapter concluded with a discussion of type conversion between numbers and strings. Implicit conversion can lead to bugs, as demonstrated, so it's best to formally convert values when needed. Along the way you also started creating practical examples, mostly as mathematical calculators.

This knowledge will be expanded in the next chapter, where you will learn about *control structures*. These are primarily conditionals and loops, but Chapter 5 will introduce more operators, too, before Chapter 6 gets into more complicated variable types.

INDEX

SYMBOLS

+ (addition) operator, 100, 102
&& (And) operator, 102, 136, 138
| (alternatives) meta-character, 407
\ (backslash), using with escape sequences, 121–122
^ (beginning of string) meta-character, 407, 411
?; (conditional) operator, 102
-- (decrement) operator, 102
/ (division) operator, 100, 102
" (double quote), using with strings, 98–99
] (end of class) meta-character, 407
} (end of quantifier) meta-character, 407
$ (end of string) meta-character, 407
) (end of subpattern) meta-character, 407
== (equal to) operator, 131
\ (escape) meta-character, 407
() (function call) operator, 102
> (greater than) operator, 102, 133
>= (greater than or equal to) operator, 102, 133
=== (identical to) operator, 131
++ (increment) operator, 102
< (less than) operator, 102, 133
<= (less than or equal to) operator, 102, 133
! (logical not) operator, 102
|| (logical or) operator, 102, 136, 138
[] (member) operator, 102
% (modulus) operator, 100, 102
* (multiplication) operator, 100, 102
! (Not) operator, 136, 138
!= (not equal to) operator, 131
!== (not identical to) operator, 131
.. (periods), using with relative paths, 38
% (remainder) operator, 100, 102
; (semicolon), using with statements, 95
. (single character) meta-character, 407
' (single quote), using with strings, 4, 98–99
// (slashes), using with comments, 99, 132
[(start of class) meta-character, 407
{ (start of quantifier) meta-character, 407
((start of subpattern) meta-character, 407
- (subtraction) operator, 100, 102
- (unary negative) operator, 104
+ (unary positive) operator, 102
_ (underscore), using with variables, 97

A

absolute vs. relative paths, 38
accessible pop-up, creating, 323–324, 338–339
ActionScript, 5–6, 20
addEvent() function, defining, 275–277
addEventListener() method, 272, 274
addition (+) operator, 100, 102
addTask() function, using with arrays, 196
addToSomething() function, 325
Adobe BrowserLab Web site, 75
Adobe Dreamweaver IDE, 67–68
Ajax
    append() method, 438
    asynchronous requests, 430
    client-side JavaScript, 12
    contact form, 456–460
    FormData object, 438
    GET method, 437–438
    GET request, 429
    impact on JavaScript, 7–13
    incorporating, 12
    JSON data, 444–447
    link click handler, 463
    login example, 453–456
    maintaining state, 457
    making requests, 429–431
    onclick anonymous function, 464
    onload anonymous function, 464
    onreadystatechange function, 435–436
    open() method for requests, 430
    overview, 426
    performing in jQuery framework, 501–502
    performing in YUI framework, 515
    popularity of, 12
    POST request, 429, 437–438
    preloading data, 461–465
    progressive enhancement, 427
    readyState property, 431–432, 434
    registration form example, 8–9
    result handler, 429
    send() method, 438
    sending data, 436–439
    sending files, 453
    server HTTP codes, 432
    server response, 431–436
    server-side requests, 12–13
    server-side script, 447–450
    statusText property, 433
    stock quotes with timer, 465–469
    synchronous requests, 430
    testing, 434
    URLs (Uniform Resource Locators), 430
    XML data, 442–444
    XMLHttpRequest object, 428–429
Ajax debugging, 439–441
    disabling cache, 441
    network monitor, 440
    PHP script, 439

Ajax object
  creating, 426, 428–429, 434
  creating for login form, 453–454
  setRequestHeader() method, 437
Ajax processes
  delaying, 452
  progress event, 452
  showing progress, 451–453
  starting, 451–452
Ajax request, invoking, 460
Ajax resources
  bidding script, 570–571
  creating for auction site, 569–572
  get bids script, 571–572
  login script, 570
ajax.js script, 457–460
  including in login.html file, 453
  try...catch block example, 478–479
Ajaxload Web site, 451
alert() method
  using, 310
  using in debugging, 85
  using wit arrays, 192
altKey property, 297
anonymous functions, using, 257–258, 260–261
Apple Safari browser. *See* Safari browser
Aptana Studio IDE, 68
arithmetic operators. *See also* assignment operators
  + (addition), 100, 102
  comparison, 104
  ?; (conditional), 102
  -- (decrement), 102
  / (division), 100, 102
  () (function call), 102
  > (greater than), 102
  >= (greater than or equal to), 102
  ++ (increment), 102
  < (less than), 102
  <= (less than or equal to), 102
  && (logical and), 102
  ! (logical not), 102
  || (logical or), 102
  [] (member), 102
  % (modulus), 100, 102
  * (multiplication), 100, 102
  new, 102
  order of precedence, 101
  % (remainder), 100, 102
  - (subtraction), 100, 102
  typeof void delete, 102
  unary, 101–102, 104
array elements
  accessing, 192–195, 198–199
  removing, 200
array functions. *See also* functions
  addTask(), 196
  alert(), 192
  concat(), 201–202
  console.log(), 192
  every(), 248–249
  filter(), 249
  forEach(), 248
  indexOf(), 194
  join(), 206
  lastIndexOf(), 194
  LIFO (Last-In, First-Out) data type, 202
  map(), 249, 252
  pop(), 202
  push(), 200
  queues, 202
  reduce(), 249
  shift(), 202
  slice() method, 203–204
  some(), 248
  splice(), 202–203
  split(), 207
  stacks, 202
  unshift(), 200
array notation, using with object properties, 210
arrays
  converting between strings, 206–207
  converting strings to, 207, 252
  converting to strings, 206
  creating, 190–192
  event listener, 197–198
  global variable, 196
  indexes, 193
  inner, 201
  length property, 194
  literal syntax, 191, 194
  multidimensional, 201
  new operator, 190–191
  .vs objects, 216–217
  in operator, 199
  passing by reference, 231
  sorting with user-defined functions, 251–253
  sparsely populated, 199
  to-do list, 195–198
  updating To-Do Manager, 204–206
assertion methods
  creating, 479–481
  using in jsUnity, 483
assignment operators, 98, 100, 102, 104. *See also* arithmetic operators
  bugs caused by, 134
  using, 100–101
asynchronous events, handling, 8
attachEvent() method, 274–275
auction site. *See also* JavaScript for auction site; PHP for auction site
  Ajax resources, 569–572
  configuration file, 558–559
  database, 556–557
  encrypting passwords, 557
  establishing, 558–559
  index.php page, 554–555
  login.php page, 554–555
  PHP scripts, 558–559

SHA1() function, 557
structure, 558
view.php page, 554–555
autocomplete, implementing, 388–389

B

Back button, linking text to, 343–344
backslash (\), using with escape sequences, 121–122
BBEdit text editor, 67
Blackbird library, 523
Boolean values, using with variables, 99
boundaries, using, 414
branching statements, 130
break control statement, using, 167
breakpoints, using in Firebug, 88
Brosera Web site, 76
browser events, 284–285
copy, 285
cut, 285
paste, 285
resize, 285
unload, 284–285
browser improvements, 14–15
browser mode, confirming, 30
browser support, 22–23
browser window, moving, 316–317
BrowserCam Web site, 76
BrowserLab Web site, 75
browserling Web site, 76
browsers
Apple Safari, 73–75
Chrome, 15, 69–70, 90
as development tools, 69
elements, 314
Firefox, 15, 69, 71, 90
hash example, 330
history property, 326–328
inner height, 314
Internet Explorer (IE), 15, 69, 72
mobile usage, 69
"modern," 22
object detection, 75
online services for testing, 76
Opera, 69, 72–73, 90
outer height, 314
outer width, 314
print option, 333
redirecting, 329–330
Safari, 15, 69
same origin security, 327
Spoon software, 76
statistics, 69
status bar, 314
testing on, 75–77
toolbar, 314
using virtualization software, 76
window.location property, 330
Browsershots Web site, 75
bugs
caused by assignment operator, 134
occurrence of, 5–6

C

calculate() function
creating for switch conditional, 147
defining, 104
calculation, performing, 100–101
calculators. *See also* numbers
creating, 103–107
discounts, 105
event listener, 106–107
init() function, 106
references to form values, 104
returning false, 106
storing order total, 104
with switch conditional, 146–150
tax rates, 105
total calculation, 105
calendar, date-picking, 15
camel-case
use in OOP, 5
using with variables, 97
Cascading Style Sheets (CSS). *See* CSS (Cascading Style Sheets)
case of characters, changing, 118
catching errors, 474–476
<![CDATA[ ]]> wrapper, using with script, 39
change events, handling, 287
character classes
[] (square brackets), 411
[0-9], 413
[^0-9], 413
[A-Za-z0-9_], 413
[^A-Za-z0-9_], 413
boundaries, 414
[f\r\t\n\v], 413
[^f\r\t\n\v], 413
meta-characters, 411
using, 411–414
characters, referencing in strings, 113. *See also* meta-characters
charAt() method, using with strings, 113
checkboxes
creating, 396–399
on e-commerce sites, 397
taking action, 397
value property, 396
Chrome browser, 15. *See also* Google
extensions, 70
features, 70
Firebug extension, 70
JavaScript Tester extension, 70
Pendule extension, 70
Speed Tracer extension, 70
usage statistic, 69
Validity extension, 70
Web Developer extension, 70
Web site, 90
circle, calculating area of, 109–110

class, start and end of, 407
client-server model, registration form in, 9
ClosureCompiler, 549
closures
- creating, 542
- creating faders, 545–546
- functions returning functions, 544
- using, 541–546

Cloud Testing Web site, 76
code, downloading, 44
code minification, 548–549
- ClosureCompiler, 549
- JSMin command-line tool, 549
- YUI Compressor, 549

comments, creating, 99, 132
comparison operators, 104, 133–136
- equal vs. identical values, 135–136
- == (equal to), 131
- > (greater than), 133
- >= (greater than or equal to), 133
- === (identical to), 131
- < (less than), 133
- <= (less than or equal to), 133
- != (not equal to), 131
- !== (not identical to), 131
- TRUE vs. FALSE conditions, 135–136

`concat()` method
- using with arrays, 201–202
- using with strings, 118

concatenating strings, 118
conditional breakpoints, setting in Firebug, 89
conditional operator, using, 102, 150–151
conditionals. *See also* `switch` conditionals
- branching statements, 130
- checking for positive radius value, 138–139
- comparing numbers, 153–154
- comparing strings, 155–159
- comparison operators, 133–136
- cryptic, 150–152
- `else` clause, 140
- `if`, 130–131
- `if-else`, 140
- `if-else if`, 141
- logical operators, 136–138
- nesting, 142
- *and* and *or* operators, 151–152
- `switch`, 143–150
- TRUE vs. FALSE, 131, 133
- `typeof` operator, 159–160
- using, 138–139

configuration object, creating, 529
`confirm()` function, using, 311
console, writing messages to, 85
`console.log()` method, using with arrays, 192
`console.trace()` function, using in debugging, 86
constants
- creating, 118
- using with `Math` object, 109

contact form. *See also* forms
- creating in Ajax, 456–460
- processing, 157–159

`contact.js` file, 157–159, 457–460
`content.js` file, creating, 461
`content.php` script, creating, 465
context and `this` variable, 254–257
control statements
- `break`, 167
- `return`, 167

control structures
- nesting, 142
- using, 131

cookie library
- creating, 361–364
- using, 364–368

cookies
- click handlers, 367–368
- contents, 358
- creating, 359–360
- deleting, 361
- expiration date and time, 358
- limitations, 359
- *name=value* pairs, 360
- overview, 358–359
- reading, 360
- retrieving, 360
- separating, 360
- setting, 367
- using, 365–368

`cookies.js` file, 361–364
Coordinated Universal Time (UTC), 180–181
Crockford, Douglas, 90, 444
CrossBrowserTesting Web site, 76
CSS (Cascading Style Sheets)
- creating modal windows, 351–356
- customizing, 365–368
- `display` property, 350–351
- hiding elements, 350–351
- and HTML vs. JavaScript, 18
- modal windows, 351–356
- referencing style sheets, 356–357
- showing elements, 350–351
- `style` property, 349–350
- `visibility` property, 350–351

CSS selectors
- `querySelector()` method, 342
- `querySelectorAll()` method, 342
- using, 341–342
- using with jQuery, 496–497
- .vs XPath expressions, 341

`ctrlKey` property, 297
cursor and mouse properties, 297
custom objects. *See also* objects
- completing, 533
- configuration object, 529
- creating, 530
- creating and using, 534–537
- multiple instances, 530–532

passing to functions, 532
tasks management application, 534–537
toString() method, 533, 535
valueOf() method, 533

D

data, preloading in Ajax, 461–465
date and time, showing, 178–180
date arithmetic
calculating intervals, 185, 188
getX(), 184–185
setX(), 184–185
timestamps, 182–184
date methods
atomic value retrieval, 176
get*() and to*(), 176–177
getTime(), 175
using, 175–180
Date objects, creating, 172, 174, 178, 180, 187
date-picking calendar, 15
dates
atomic values, 173
changing, 181–182
creating, 172–175
errors as messages, 189
event listeners, 189
process() function, 186
RFC822/IETF format, 175
set*() methods, 181–182
start and end for events, 186–190
using strings, 174–175
using timestamps, 174
validating for events, 187
debugging. *See also* Firebug
Ajax, 439–441
JavaScript, 17
with text editor vs. IDE, 63
debugging techniques
alert() method, 85
browser console, 83
browsers, 84
coding, 85
console.trace() function, 86
development browser, 83
external files, 84
IDEs (Integrated Development Environments), 83
JavaScript validator, 83
log() method, 85
network monitor, 86
rubber duck, 84
saving and refreshing, 84
text editors, 83
writing messages to console, 85
decrement (--) operator, 102
default behavior, preventing, 297–301
development approaches
graceful degradation, 39–41
noscript element, 39–41
progressive enhancement, 41–42, 45
unobtrusive JavaScript, 43, 52
dialog windows. *See also* windows
alerts, 310
confirmations, 311
customizing, 312
\n (newline) character, 312
prompts, 312
using, 310–312
discount, including in calculator, 105
division (/) operator, 100, 102
do...while loop, using, 166
DOCTYPE
benefits, 30
choosing, 28–30
HTML 4.01, 28
Transitional option, 28–30
triggering Quirks mode, 30
XHTML 1.0, 28
document, requesting from server, 48
document object
using, 333–334
write() method, 333–334
writeln() method, 333–334
document.compatMode, 334
document.createElement() method, 344
The Dojo Toolkit framework, 16
DOM (Document Object Model), 29
adding elements to, 345
changing elements, 342–344
copying elements, 346
creating, 48
creating elements, 344–348
creating print button, 347–348
CSS selectors, 341–342
Level 0 specification, 272
Level 2 specification, 271, 273
manipulation, 338–339
nodes, 336–337
nodeType property, 337
overview, 335–337
removeChild() method, 346
replacing elements, 345
shortcuts, 337–338
tree representation, 335–336
DOM elements, referencing, 48
DOM methods, 340
dot notation, chaining, 5
double quote ("), using with strings, 98–99
Dreamweaver IDE, 67–68
duck typing, using to test value types, 548
dynamically typed language, 6

E

Eclipse IDE, 68
ECMAScript, 6, 22
EditPlus text editor, 66
Edwards, Dean, 90
Eich, Brendon, 90

else clause, using, 140
Emacs text editor, 67
email address, validating, 414
employee.html page, 212–213
employee.js file
  creating, 213
  opening, 256
  saving, 215, 257
epoch.js file, creating, 280
equal to (==) operator, 131
error causes
  = instead of ==, 82
  angle brackets, 82
  curly braces, 82
  function names, 81
  object names, 81
  object references, 82
  object types, 82
  parentheses, 82
  quotation marks, 82
  reserved words, 82
  variable names, 81
error management
  assertions, 479–481
  unit testing, 481–485
error messages
  adding, 380–383
  creating for forms, 379–383
  removing, 380–383
  span, 381
error types
  logical, 80–81
  run-time, 80–81
  syntactical, 80
errorMessages.js file
  creating, 380
  saving, 383
errors
  catching, 474–476
  finally clause, 476
  in try block, 475
escape (\) meta-character, 407
escape sequences, 121–122
eval() function, using with windows, 371
event assigner, creating, 273–274
event handlers
  inline, 269, 272
  naming, 270
event handling
  delegating, 304
  event phases, 302–304
  event properties, 291–295
  finding key pressed, 296–297
  IE (Internet Explorer), 273
  preventing default behavior, 297–301
  progressive enhancement, 269
  referencing events, 290–291
  traditional, 269–272
  W3C (World Wide Web Consortium), 271–273
event listeners
  addEventListener() method, 272
  adding for dates, 189
  adding to calculator, 106–107
  adding to forms, 46–47, 49–50, 118, 121
  adding to page elements, 274
  adding to random.js file, 165
  creating, 268–274
  using with arrays, 197–198
  using with objects, 215
event phases
  advantages, 304
  bubbling, 302–303
  capturing, 302–303
  relatedTarget property, 304
event types
  browsers, 284–285
  forms, 286–287
  input devices, 278–282
  keyboards, 282–284
event-driven language, explained, 46
event.js file, 186–190
events
  accessibility, 287–288
  associating with functions, 268
  asynchronous, 8
  handling, 46–50
  pairing, 288
  progressive enhancement, 288–289
  reliability, 287
  reporting on, 292–295
  this variable, 295
events.html page, 292
events.js file, creating, 292
every() array function, 248–249
exceptions, throwing, 475, 477–478
execution context and this variable, 254–257
expressions .vs statements, 245
Extensible Markup Language (XML). *See* XML (eXtensible Markup Language)
ExtJS framework, 16

F

fader, creating with closure, 545–546
fallthroughs, performing, 144
FALSE
  determining for control structures, 131, 133
  vs. TRUE conditions, 135
false and true values, 99
file uploads, handling, 401–402
filter() array function, 249
finally clause, adding to try...catch, 476
Firebug. *See also* debugging
  applying to Web pages, 87
  assertions in, 481
  breakpoints, 88–89
  clear() function, 87

conditional breakpoints, 89
Console tab, 87
Continue in Script panel, 88–89
executing lines of JavaScript, 87
`inspect()` function, 87
opening, 87
Rerun in Script panel, 88–89
Script panel for debugging, 88
Step Into in Script panel, 88–89
Step Out in Script panel, 88–89
Step Over in Script panel, 88–89
using, 86–89
watch expressions, 89–90
Wiki, 89
Firefox browser, 6, 15
Console2 extension, 71
extensions, 71
features, 71
Firebug extension, 71
Greasemonkey extension, 71
JS View extension, 71
Total Validator extension, 71
usage statistic, 69
View Source Chart extension, 71
Web Developer extension, 71
Web site, 90
YSlow! extension, 71
Flash vs. JavaScript, 20
focus, changing, 321
`for` loop
defining in `random.js` file, 163
executing, 161–162
program flow, 161
syntax, 161–162
using, 163
using with arrays, 201
`for...in` loop, using with object properties, 211
`forEach()` array function, 248
form data, problems with, 8
form events
blur, 286
change, 286–287
focus, 286
reset, 286
select, 286
form input, assigning values to, 106
form submission
handling, 378
preventing default behavior, 378–379
forms. *See also* contact form; login form
accessibility, 378
`action` attribute, 378
autocomplete, 388–389
baseline functionality for, 42
checkboxes, 396–399
client-side validation, 9–10
disabling submit button, 386
error messages, 379–383
file uploads, 401–402
preventing submission of, 106
radio buttons, 400–401
`register.js` example, 416–420
registration page example, 415–420
select menus, 389–396
server-side validation, 10–11
text inputs, 387–388
textareas, 387–388
tooltips, 383–385
validation, 379
frames, `iframe`, 328
frameworks, 15–16. *See also* jQuery framework; YUI framework
arguments against use of, 16
choosing, 16
considering, 493
The Dojo Toolkit, 16
ExtJS framework, 16
jQuery, 16
MooTools, 16
overview, 492, 494
Prototype, 16
script.aculo.us, 16
YUI (Yahoo! User Interface), 16
Fuchs, Thomas, 90
function call `(())` operator, 102
function keyword, using, 50
function parameters, 226, 228–229, 241–242
functionality, developing, 44–45
functions. *See also* array functions
anonymous, 257–258, 260–261
applying to variables, 4
as argument values, 246–248
`arguments` variable, 227
associating events with, 268
context and `this` variable, 254–257
creating and calling, 232–234, 236–238
defined, 4, 49
defining, 222–223
design theory, 243
immediately invoked, 257–261
lack of default values, 228–229
lack of parameter checking, 228
lack of type checking, 226
local scope, 239
nested, 258–261
as objects, 244–248
passing objects to, 231
passing values, 230–234
passing values to, 223–225
recursion, 261–262
returning objects, 235
returning values from, 234–238
`sort()` method, 246–248
user-defined, 251–253
variable scope, 238–243
as variable values, 245–246

G

getElementById() method, using with form, 47
getRandomNumber() function, calling, 238
getTime(), using with dates, 175
getTimeZoneOffset() method, 181
getX(), using with dates, 184–185
Git version control software, 62
global variables, 95. *See also* variables
  in functions, 239–240
  namespace pollution, 243
  problem with, 243
  using with arrays, 196
GMT (Greenwich Mean Time), 180
Google, browser support, 22. *See also* Chrome browser
graceful degradation, 39–41
Graded Browser Support, 23
greater than (>) operator, 102, 133
greater than or equal to (>=) operator, 102, 133

H

handling events, 46–50
hash example, 330
hash property, 330–332
hash value, watching for changes in, 372
Head JS library, 522–523
Heilmann, Christian, 90
history property
  back() method, 326–327
  forward() method, 326–327
  go() method, 326–327
HTML (HyperText Markup Language)
  avoiding use of dummy links, 43
  and CSS vs. JavaScript, 18
  DOCTYPE, 28
  Semantic, 41–42
  vs. XHTML, 28
HTML buttons, using, 289
HTML document, loading, 48
HTML elements
  adding to DOM, 345
  changing, 343
  cloneNode() method, 346
  copying, 346
  creating, 344–348
  customizing, 344
  innerHTML property, 343
  placing text in, 163
  replacing, 345
HTML forms
  example of, 8
  validating, 46
HTML pages
  adding JavaScript to, 37–39
  path/to part, 37
  script element, 37
  testing looks of, 75
  tree representation, 335
  validating, 28
HTML5
  explained, 31
  form elements, 34–35
  pattern attribute, 36
  template, 31–33
  vs. XHTML, 36

I

identical to (===) operator, 131
IDEs (Integrated Development Environments)
  Adobe Dreamweaver, 67–68
  Aptana Studio, 68
  Eclipse, 68
  features, 65
  IntelliJ IDEA, 68
  JetBrains, 68
  Komodo IDE, 67
  NetBeans, 68
  PhpStorm, 68
  price range, 64
  WebStorm, 68
IE versions, testing HTML pages on, 76
if conditional
  FALSE, 130
  omitting curly braces ({}), 130
  syntax, 130–131
  TRUE, 130
if-else conditionals, using, 140
if-else if conditionals, using, 141
iframe, using, 328
images, preloading in Ajax, 464
in operator
  using with arrays, 199
  using with object properties, 210
increment (++) operator, 102
indexes
  using with arrays, 193
  using with methods for strings, 113
indexOf() method
  using with arrays, 194
  using with strings, 114, 155, 408
index.php page, in auction site, 554–555
Infinity value, returning, 102
init() function
  calling, 49–50, 52–53
  using with calculators, 106
innerText property, using, 163
input device events, 278–282
  click, 278
  contextmenu, 279
  double-click, 279
  input button, 278–279
  input movement, 279–282
  mousedown, 278
  mousemove, 279
  mouseout, 279
  mouseover, 279–282
  mouseup, 278
  touch devices, 281

Integrated Development Environments (IDEs). *See* IDEs (Integrated Development Environments)
IntelliJ IDEA IDE, 68
Internet Explorer (IE) browser, 15
  event handling, 273
  features, 72
  usage statistic, 69
intervals, calculating for dates, 185, 188
Irish, Paul, 90
`isFinite()` function, using with numbers, 154
`isNan()` function, using with numbers, 154
iteration .vs recursion, 262

J

JavaScript
  vs. ActionScript, 20
  adding to HTML pages, 37–39
  benefits, 21
  browser improvements, 14–15
  browser support, 22–23
  case-sensitivity of, 81
  current version of, 22
  debugging, 17
  dynamically typed, 6
  ECMAScript implementation, 6
  execution of, 40
  features, 17
  vs. Flash, 20
  founders, 90
  frameworks, 15–16
  as Good Thing, 21
  vs. HTML and CSS, 18
  impact of Ajax, 7–13
  vs. Java, 17
  learning curve, 14, 17
  as object-oriented language, 4–5
  original uses of, 7
  overview, 4–6
  vs. PHP, 18–19
  programming goals, 24–25
  progressive enhancement, 24
  prototype-based, 5
  putting between in `script` tags, 43
  scripting language, 6
  security concern, 17
  testing, 77–79
  unobtrusive, 24, 43
  versions, 22–23
  weakly-typed, 5, 95
JavaScript 1.0, release of, 6
JavaScript alert, appearance of, 52
JavaScript code, executing, 77
JavaScript for auction site. *See also* auction site; `view.js` file for auction site
  completing, 592
  `login.js` file, 572–578
  `utilities.js` file, 572
JavaScript layer, adding, 45–46
JetBrains IDEs, 68
`join()` method, using with arrays and strings, 206
jQuery framework, 16. *See also* frameworks
  CDN (Content Delivery Network) version, 495
  changing CSS classes, 498
  creating effects, 501
  CSS selectors, 496–497
  DOM manipulation, 498–499
  downloading, 494
  features, 494
  handling events, 500–501
  manipulating elements, 497–498
  performing Ajax, 501–502
  selecting page elements, 496–497
  UI library, 503
  using, 495–496
`jQuery()` function, using, 495
jQuery Mobile library, 523
jQuery plug-ins
  Autocomplete widget, 504–507
  DataTables, 507–508
  date-picker widget, 503
  using, 503–504
JS Bin tool
  keyboard shortcuts, 79
  using, 78–79
JScript implementation, 6
jsFiddle Web site, 79
JSHint validator, using, 83
JSLint validation service, using, 83
JSMin command-line tool, 549
JSON data
  returning, 450
  sending to server, 445
  using with Ajax, 444–447
  validating, 440
jsUnity library
  assertion methods, 483
  using in unit testing, 482

K

key pressed, finding, 296–297
keyboard events, 282–284
  handling, 283–284
  keydown, 282
  keypress, 282
  keyup, 282
Komodo Edit text editor, 66
Komodo IDE, 67

L

`lastIndexOf()` method
  using with arrays, 194
  using with strings, 114–115
`length` property, using with arrays, 194
less than (<) operator, 102, 133

less than or equal to (<=) operator, 102, 133
libraries
Blackbird, 523
Head JS, 522–523
jQuery Mobile, 523
MediaElement.js, 523
Modernizr, 522
RequireJS, 523
Sencha Touch, 523
SWFObject, 522
VideoJS, 523
Zepto, 523
LIFO (Last-In, First-Out) data type, 202
literal syntax, using with arrays, 191, 194
literals vs. objects, 94, 124
local scope, explained, 239
`log()` method, using in debugging, 85
logical operators, 102, 136–138
login form. *See also* forms
adding JavaScript layer, 45–46
base functionality, 44–45
`getElementById()` method, 47
`init()` function, 49, 52–53
JavaScript alert, 52
submission event, 47
`validateForm()` function, 52–53
validating, 50–54
`loginForm` object, `onsubmit` property, 49
`login.html` file, 44
including `ajax.js` script in, 453
`readyState` change handling function, 454–455
`login.js` file, 45, 54
creating Ajax object, 453–454
saving, 455
writing for auction site, 573–578
`login.php` script
in auction site, 554–555
creating, 455–456
submitting login form to, 45
loops
`do...while`, 166
`for`, 161–165
`for...in`, 211
nesting, 166
`while`, 166

M

MAMP for Mac OS X, 430
`map()` array function, 249, 252
math, performing with strings, 123
`Math` object
`abs()` method, 110
`ceil()` method, 110
constants, 109
`cos()` method, 110
`floor()` method, 110
`max()` method, 110
`min()` method, 110
`pow()` method, 110
predefined methods, 110
`random()` method, 110
`round()` method, 110
`sin()` method, 110
using, 109–112
MediaElement.js library, 523
member ([ ]) operator, 102
membership cost calculation, 299
`membership.html` file, using, 145–150
`membership.js` file, preventing default behavior, 300–301
meta-characters. *See also* characters
| (alternatives), 407
^ (beginning of string), 407, 411
in character classes, 411
] (end of class), 407
} (end of quantifier), 407
$ (end of string), 407
) (end of subpattern), 407
\ (escape), 407
. (single character), 407
[(start of class), 407
((start of subpattern), 407
{ (start of quantifier), 407
using with patterns, 406–407
methods. *See* functions
Microjs Web site, 523
Minify JavaScript Web Site, 548
mobile browsers, usage of, 69
modal windows, creating, 351–356. *See also* windows
`modal.css` file, 353–355
`modal.html` file, 351–353
`modal.js` file
`closeModal()` function, 355
creating, 355
`openModal()` function, 355
saving, 356
Modernizr library, 522
modulus (%) operator, 100, 102
Mogotest Web site, 76
MooTools framework, 16
mouse and cursor properties, 297
mouseover event, handling, 280–282
Mozilla Firefox. *See* Firefox browser
multiplication (*)operator, 100, 102

N

`\n` (newline) character, using with dialogs, 312
namespace pollution, 243
namespaces, defining, 528–529
`NaN` value, returning, 102
nested functions, using, 258–261
nesting
conditionals, 142
control structures, 142
loops, 166
NetBeans IDE, 68
network monitor
for Ajax debugging, 440
using, 63
using in debugging, 86

new operator
  explained, 102
  using with arrays, 190–191
noscript element, using, 39–41
not equal to (!=) operator, 131
not identical to (!==) operator, 131
Notepad text editor, 66
null value, using, 99, 125
Number object type, 108
numbers. *See also* calculators
  adding to strings, 122–123
  arithmetic operators, 100–103
  comparing, 153–154
  comparing to strings, 156
  converting strings to, 123–124
  creating years to, 147
  formatting, 107–109
  Infinity value, 102
  isFinite() function, 154
  isNan() function, 154
  NaN value, 102
  toFixed() method, 108
  toPrecision() method, 108

O

object detection, using, 42, 50, 75
object event properties, 49
object inspectors, using, 211
object methods
  creating, 256–257
  using this keyword with, 256
object notation, using, 4
object properties
  accessing, 209–211
  array notation, 210
  creating, 208
  events, 49
  for...in loop, 211
  in operator, 210
  removing, 212–215
  testing for, 210
  typeof operator, 211
object-oriented language, 4–5
objects. *See also* custom objects
  .vs arrays, 216–217
  associating with functions, 268
  components of, 207
  creating, 207–209
  event listener, 215
  functions as, 244–248
  vs. literals, 94, 124
  mutable and immutable, 212
  passing by reference, 231
  passing to functions, 231
  process() function, 213
  returning from functions, 235
  using, 213–215
Opera browser, 72–73
  Dragonfly development tool, 73
  usage statistic, 69
  Web site, 90
order of precedence, 101, 137
os.js file, 392–396

P

parent directory, moving up to, 38
parseFloat() method, 123, 125
parseInt() method, 123, 125
passing
  by reference, 230
  by value, 230
passwords, encrypting, 557
paths, absolute vs. relative, 38
patterns
  defining, 406–408
  literals, 406
  meta-characters, 406–407
  using, 408, 410
periods (..), using with relative paths, 38
phone number, validating, 418
PHP
  vs. JavaScript, 18–19
  Web site, 90
PHP for auction site. *See also* auction site
  bid form submission, 568–569
  creating bid form, 567–568
  current bids, 569
  displaying item details, 565–566
  listing auctions, 560–563
  logging in, 563–564
  validating item ID, 564–565
  viewing auctions, 564–569
PhpStorm IDE, 68
pizza.js file, checkbox example, 398–399
plain text, returning, 447–448. *See also* text
pop() method, using with arrays, 202
popup.js file
  creating, 323
  opening, 338
  saving, 324, 339
pop-ups
  accessible solution, 323–324, 338–339
  customizing, 319–321
postfix vs. prefix versions, 101
prefix vs. postfix versions, 101
preloading
  data in Ajax, 461–465
  images in Ajax, 464
print button, creating, 347–348
printing pages, 333
procedural language vs. object-oriented language, 4
process() function, using with objects, 213
progressive enhancement
  Ajax, 427
  and events, 288–289
  explained, 24
  limitation, 269
  overview, 41–42
prompt() function, using, 312
properties, defined, 4

Prototype framework, 16
prototype-based language, 5
prototypes
  changing, 540
  inheritance, 538
  methods, 539–540
  overview, 537
  trim() method, 539
push() method, using with arrays, 200

Q

quantifiers, 409
Quirks mode, triggering, 30
quotation marks, using with variables, 98–99
quote.js file, 466–468
quote.php script, creating, 468

R

radio buttons
  dynamic effects, 401
  flag variable, 400
  using, 400–401
random numbers
  generating, 164–165
  returning, 237
random.html page, 163–165
random.js file
  creating, 164
  saving, 165
  showNumbers() function, 164
recursion
  .vs iteration, 262
  performing, 261–262
reduce() array function, 249
register.js file, 416–420
registration form example, 8–9, 415–420
regular expressions
  creating, 404
  defining patterns, 406–408
  exec() function, 405–406
  functions, 405–406
  literals, 406
  match() function, 405–406
  meta-characters, 406–407
  overview, 403–404
  performance issues, 412
  RegExp object type, 404
  replace() method, 406
  rules for, 411
  search() function, 405
  split() method, 406
  test() function, 405
relatedTarget property, 304
relative vs. absolute paths, 38
remainder (%) operator, 100, 102
removeEvent() method, defining, 277
reportEvent() function, creating, 292
RequireJS library, 523
Resig, John, 90
resize event, triggering, 285
return statement, using, 167, 236
RFC822/IETF format, using with dates, 175
RIAs (Rich Internet Applications), 20
Ruby Web site, 90

S

Safari browser, 15, 73–75
  Develop menu, 74
  disabling JavaScript, 74
  usage statistic, 69
  Web Inspector, 74
Sauce Labs Web site, 76
screen properties, using with windows, 317
script element
  <![CDATA[ ]]> wrapper, 39
  parsing data in, 39
  using, 37–39
script tags, putting JavaScript between, 43
script.aculo.us framework, 16
scripting language, JavaScript as, 6
scripts, organizing, 107
select menus
  creating, 389–390
  dynamic select boxes, 390–396
  linking, 392–396
  validating, 390
Semantic HTML, using, 41–42
semicolon (;), using with statements, 95
Sencha Touch library, 523
server-side requests, 12–13
server-side script
  returning JSON, 450
  returning plain text, 447–448
  returning XML, 449–450
server-side validation, 10–11
setHandlers() function, defining, 293
setText() function
  defining, 237, 251
  for utility library, 276
setX(), using with dates, 184–185
Sexton, Alex, 90
SHA1() function, using with passwords, 557
Sharp, Remy, 78, 90
shift() method, using with arrays, 202
shiftKey property, 297
shopping.html page, creating, 103
shopping.js file, 107–109
single quote ('), using with strings, 4, 98–99
slashes (//), using with comments, 99, 132
slice() method
  using with arrays, 203–204
  using with strings, 115–116
some() array function, 248
sort() method, using with functions, 246–248

sortWords() function
completing, 253
defining, 252
span, adding to DOM for errors, 381
sphere, calculating volume of, 111–112
sphere.js file, 110, 138–139
splice() method, using with arrays, 202–203
split() method, using with arrays and strings, 207
Spoon software, using, 76
srcElement event property, 291–292
state, maintaining in Ajax, 457
statements .vs expressions, 245
stock ticker, creating, 466–469
strict mode, invoking, 53
strings
adding to numbers, 122–123
beginning and end of, 407
changing case, 118, 155
charAt() method, 113
comparing, 155–159
comparing to numbers, 156
concatenating, 118
converting arrays to, 206
converting to arrays, 207, 252
converting to numbers, 123–124
creating, 112–113
deconstructing, 113–118
empty, 99
escape sequences, 121–122
example of, 4
indexes for methods, 113
indexOf() method, 114, 155, 408
lastIndexOf() method, 114–115
length property, 113
manipulating, 120–121
matching, 408
performing math with, 123
processing contact form, 157–159
referencing characters, 113
slice() method, 115–116
substr() method, 115
substring() method, 115
toLowerCase() function, 118, 155
toUpperCase() function, 118, 155
trim() method, 118
using with dates, 174–175
strongly typed language, 5
style sheets
addRule() method, 357
createElement() method, 357
deleteRule() method, 357
disabled property, 356
insertRule() method, 357
referencing, 356–357
submission event, watching for, 47
submit button, disabling, 386
substr() method, using with strings, 115
substring() method, using with strings, 115
subtraction (-) operator, 100, 102
Subversion version control software, 62
SWFObject library, 522
switch conditionals. *See also* conditionals
calculator, 146–150
default case, 144
expressions, 145
fallthroughs, 144
identity matches, 144
membership.html file, 145–146
parentheses in, 143
quotes in, 143
using, 143–150
syntax highlighting, 60–61

T

tasks.js file
closure example, 540
creating, 196
custom objects example, 535–537
saving, 198
tax rate, including in calculator, 105
ternary operator, using, 150–151
testing
on browsers, 75–77
JavaScript, 77–79
test.js file, 434–436
tests
creating for utilities library, 485–488
defining for unit testing, 482–483
log() function, 487
preparing for, 484–485
running, 487
running for unit testing, 483–484
setUp() function, 484–485
tests.js file, 485–488
text, placing in HTML elements, 163. *See also* plain text
text editor vs. IDE (Integrated Development Environment)
choosing between, 66
code completion, 61
code intelligence, 61–62
common features, 60–64
comparing, 64–65
debugging, 63
file management, 62
HTML and CSS, 64
network monitor, 63
project management, 62
syntax highlighting, 60–61
unit testing, 63
version control software, 62
vi editor, 64
text editors, 64–65
BBEdit, 67
EditPlus, 66
Emacs, 67
features, 65
hardware resources, 64
Komodo Edit, 66
Notepad, 66
price range, 64

text editors (*continued*)
TextMate, 66
TextWrangler, 66
UltraEdit, 66
Vim, 67
text form input, assigning values to, 106
text inputs
retrieving contents of, 387
`value` attribute, 387
textareas
retrieving contents of, 387
`value` attribute, 387
`textContent` property, using, 163
`text.html` page, 116–117
`text.js` file
for change events, 287
creating, 283
saving, 284, 287
using, 116–118
TextMate text editor, 66
TextWrangler text editor, 66
`theme.js` file, using with cookies and CSS, 365–368
`this` variable
using with context, 254–257
using with events, 295
throwing exceptions, 477–478
time and date, showing, 178–180
time zones
`getTimeZoneOffset()` method, 181
using, 180–181
timers
changes in hash values, 372
`clearInterval()` function, 369–370
`clearTimeout()` function, 370
`setInterval()` function, 369
`setTimeout()` function, 369–370
stock quotes example, 465–469
using, 369–372
times, creating dates for, 172–173
timestamps, using, 174
`today.js` file
creating, 178
opening, 232
saving and testing, 234
to-do list, creating with arrays, 195–198
To-Do Manager, updating, 204–206
`toLowerCase()` function, using with strings, 155
tooltips
creating, 383–385
`hideTooltip()` function, 385
hiding, 384
`showTooltip()` function, 385
`visibility` property, 384
`toString()` method, using in type conversions, 125
touch devices, input events, 281
`toUpperCase()` function, using with strings, 155
tree structure, 288–289
`trim()` method
using with prototype, 539
using with strings, 118
trinary operator, using, 150–151
TRUE
determining for control structures, 131, 133
vs. FALSE conditions, 135
`true` and `false` values, 99
`try` block
errors in, 475
throwing exception in, 477–478
`try...catch` block
`finally` clause, 476
syntax, 474
using, 478–479
type conversions
`parseFloat()` method, 123, 125
`parseInt()` method, 123, 125
performing, 122–125
`toString()` method, 125
type identification, alternative, 547–548
`typeof` operator
alternative, 547–548
`Array` type, 159–160
`Boolean` type, 159
`NaN` value, 159
`Null` type, 159–160
`Number` type, 159
`Object` type, 159
return values, 159
`String` type, 159
`Undefined` type, 159
using, 159–160
using with object properties, 211
`typeof void delete` operator, 102

U

U object
creating, 275
finishing declaration of, 277
UltraEdit text editor, 66
unary operators, 101–102, 104
`undefined` value, using, 99, 125
underscore (_), using with variables, 97
Unicode character, returning, 296
unit testing, 481–482
defining tests, 482–483
logging results, 484
on multiple browsers, 487
performing, 485–488
setting up jsUnity library, 482
support for, 63
unload event, triggering, 284
unobtrusive JavaScript, 43, 52
`unshift()` method, using with arrays, 200

URLs (Uniform Resource Locators)
  # part of, 331
  creating, 331–332
  deep linking, 331
  parsing hash in, 331–332
user experience, improving, 24
UTC (Coordinated Universal Time), 180–181
utilities library
  creating, 275–277
  creating unit tests for, 485–488
`utilities.js` file, 275–277

V

`validateForm()` function, using, 47, 50–53
validating
  HTML forms, 46
  HTML pages, 28–29
  JSON, 440
  phone number, 418
  XML, 440
validation, performing, 50–54
validation services, using, 83
validators, W3C Markup Validation Service, 91
`value` attribute
  using, 105–106
  using with text inputs, 387
  using with textareas, 387
value types
  Booleans, 98
  duck typing, 548
  exponential notation, 98
  `Infinity` value, 102
  `NaN` value, 102
  `null`, 99
  numbers, 98
  quotation marks, 98–99
  strings, 98
  testing, 548–549
  `true` and `false`, 99
  `undefined`, 99
values
  assigning to variables, 98
  equal vs. identical, 135
  literals .vs objects, 94
  passing to functions, 223–225
  returning from functions, 234–238
`var` keyword, using, 95–96
variable scope
  explained, 239
  function parameters, 241–242
variables. *See also* global variables
  applying functions to, 4
  camel-case, 97
  declaring, 94–96, 136
  declaring outside of functions, 96
  global scope, 95–96
  hoisting, 96
  identifiers, 97
  local .vs global, 239–240
  names, 97
  undeclared, 95
  use of underscore (_), 97
  value types, 98–99
  values, 98
version control software, using, 62
vi editor, using, 64
VideoJS library, 523
`view.js` for auction site. *See also* JavaScript for auction site
  `getBids()` function, 589
  `handleBidAjaxResponse()` function, 583
  `handleGetBidsAjaxResponse()` function, 586, 588
  `init()` function, 589, 591
  load handler, 591
  structure, 581
  `submitBid()` function, 585–586
  writing for auction site, 581–591
`view.php` page
  in auction site, 554–555
  writing for auction site, 578–581
Vim text editor, 67
virtualization software, using, 76

W

W3C event handling, 271–273
W3C Markup Validation Service, 91
watch expressions, creating in Firebug, 89–90
weakly-typed language, 5, 95
Web browsers. *See* browsers
Web sites
  Adobe BrowserLab, 75
  Adobe Dreamweaver IDE, 67–68
  Ajaxload, 451
  Apple Safari browser, 73
  Aptana Studio IDE, 68
  BBEdit, 67
  Blackbird, 523
  Brosera, 76
  BrowserCam, 76
  browserling, 76
  browsers, 90
  Browsershots, 75
  Chrome, 70
  Cloud Testing, 76
  Crockford, Douglas, 90
  CrossBrowserTesting, 76
  The Dojo Toolkit, 16
  Eclipse IDE, 68
  ECMAScript 5, 22
  EditPlus, 66
  Edwards, Dean, 90
  Eich, Brendon, 90
  Emacs, 67
  ExtJS framework, 16
  Firebug Wiki, 89
  Firefox browser, 6, 71

Web sites (*continued*)
Fuchs, Thomas, 90
Git version control software, 62
Google Chrome, 70
Graded Browser Support, 23
Head JS library, 522–523
Heilmann, Christian, 90
IntelliJ IDEA, 68
Internet Explorer, 72
Irish, Paul, 90
JetBrains IDEs, 68
jQuery framework, 16, 494
jQuery Mobile, 523
JS Bin tool, 78
jsFiddle, 79
JSHint validator, 83
JSLint validation service, 83
Komodo Edit, 66
Komodo IDE, 67
MAMP for Mac OS X, 430
MediaElement.js library, 523
Microjs, 523
Minify JavaScript, 548
Modernizr library, 522
Mogotest, 76
MooTools, 16
Mozilla Firefox browser, 6, 71
NetBeans IDE, 68
Notepad, 66
Opera browser, 72–73
PHP, 90
PhpStorm, 68
Prototype, 16
RequireJS library, 523
Resig, John, 90
Ruby, 90
Safari browser, 73
Sauce Labs, 76
script.aculo.us, 16
Sencha Touch, 523
Sexton, Alex, 90
Sharp, Remy, 90
Spoon software, 76
Subversion version control software, 62
SWFObject library, 522
TextMate, 66
TextWrangler, 66
UltraEdit, 66
validation services, 83
version control software, 62
VideoJS library, 523
Vim, 67
W3C Markup Validation Service, 28, 91
WebStorm, 68
XAMPP for Windows, 430
YUI (Yahoo! User Interface), 16
Zepto, 523
WebStorm IDE, 68
`while` loop, using, 166
`window` object
close() method, 319
focus() method, 321
global, 313–315
height property, 320
innerHeight property, 316
innerWidth property, 316
left property, 320
location property, 320
members, 314
menubar property, 320
moveTo() method, 316–317
open() method, 318–320
outerHeight property, 316, 320
outerWidth property, 316, 320
print() method, 333
properties, 315
resizable property, 320
screen properties, 317
screenX property, 316
screenY property, 316
scrollbars property, 320
status property, 320
toolbar property, 320
top property, 320
width property, 320
window.navigator property, 315
window properties, document object, 333–334
window.frames property, 328
window.history.back() method, 343–344
window.location property
hash property, 330–332
search property, 330
using with browsers, 330
windows, 371. *See also* dialog windows; modal windows
accessible solution, 322–324
addToSomething() function, 325
browser's history, 326–328
changing focus, 321
communicating between, 325–326
creating, 318–319, 322
customizing pop-ups, 319–321
document object, 333–334
eval() function, 371
global window object, 313–315
printing pages, 333
redirecting browsers, 329–330
repositioning, 315–317
representative URLs, 331–332
resizing, 315–317
screen properties, 317
target attribute, 322
words.html page, 250–251
words.js file, 251–253

X

XAMPP for Windows, 430
XHTML
  vs. HTML, 28
  vs. HTML5, 36
XML (Extensible Markup Language)
  `documentElement`, 442–443
  fetching, 442
  `getAttribute()` method, 443
  `getElementsByTagName()` method, 443
  returning, 449–450
  sending to server, 445
  using with Ajax, 442–444
  validating, 440
`XMLHttpRequest` object, 428–429
XPath expressions vs. CSS selectors, 341

Y

Yahoo!, Graded Browser Support, 23
Yahoo! Query Language (YQL) utility, using, 518–522
years, converting to numbers, 147
YQL (Yahoo! Query Language) utility, using, 518–522
YUI (Yahoo! User Interface), 15–16
YUI Compressor, 549
YUI framework. *See also* frameworks
  Autocomplete widget, 516–517
  creating effects, 514–515
  DOM manipulation, 513–514
  handling events, 514
  manipulating elements, 512–513
  overview, 509
  performing Ajax, 515
  selecting elements, 511–512
  skinning widgets, 516
  using, 509–511
  widgets and utilities, 516–522
  YQL (Yahoo! Query Language) utility, 518–522

Z

Zepto library, 523