PART

4

Appendixes

APPENDIX A:

Essential CSS

APPENDIX B:

JavaScript: The Brains of Your Page

APPENDIX

A

Essential CSS

It's no exaggeration to say that modern web design wouldn't be possible without *CSS*, the Cascading Style Sheet standard. CSS allows even the most richly formatted, graphically complex web pages to outsource the formatting work to a separate document—a *style sheet*. This keeps the web page markup clean, clear, and readable.

To get the most out of HTML5 (and this book), you need to be familiar with the CSS standard. If you're a CSS pro, don't worry about this appendix—carry on with the material in the rest of the book, and pay special attention to Chapters 6 and 7, which introduce many of the new style features that CSS3 adds. But if your CSS skills are a bit rusty, this appendix will help to refresh your memory before you go any further.

NOTE This appendix gives a very quick (and not comprehensive) rundown of CSS. If you're still overwhelmed, consult a book that deals with CSS in more detail, like *CSS3: The Missing Manual* by David Sawyer McFarland.

Adding Styles to a Web Page

There are three ways to use styles in a web page.

The first approach is to embed style information directly into an element using the `style` attribute. Here's an example that changes the color of a heading:

```
<h1 style="color: green">Inline Styles are Sloppy Styles</h1>
```

This is convenient, but it clutters the markup terribly. You have to style every line, one by one.

The second approach is to embed an entire style sheet in a <style> element, which you must place in the page's <head> section:

```
<head>
  <title>Embedded Style Sheet Test</title>
  <style>
   ...
  </style>
</head>
```

This code separates the formatting from your web page markup but still keeps everything together in one file. This approach makes sense for one-off formatting tasks (when you don't want to reuse your formatting choices in another page), and it's a good choice for simple tests and examples, like the ones included with this book. However, it's not so great for a real-world, professional website, because it leads to long, bloated pages.

The third approach is to link to a separate style sheet file by adding a <link> element to the <head> section. Here's an example that tells a web browser to apply the styles from the style sheet named *SampleStyles.css*:

```
<head>
  <title>External Style Sheet Test</title>
  <link rel="stylesheet" href="SampleStyles.css">
</head>
```

This approach is the most common and the most powerful. It gives you the flexibility to reuse your styles in other pages. If you want, you can further divide your styles into multiple style sheets and link to as many as you need in any HTML page.

NOTE A simple philosophy underpins modern web development. HTML markup is for structuring a page into logical sections (paragraphs, headings, lists, images, links, and so on), while a CSS style sheet is for formatting it (by specifying fonts, colors, borders, backgrounds, and layout). Follow this rule, and your web pages will be easy to edit. You'll also be able to change the formatting and layout of your entire website simply by modifying its linked style sheet. (To see a truly impressive example of style sheet magic, check out *www.csszengarden.com*, where one website is given more than 200 different faces, simply by swapping in different style sheets.)

The Anatomy of a Style Sheet

A style sheet is a text file, which you'll usually place on a web server alongside your HTML pages. It contains one or more *rules*. The order of these rules doesn't matter.

Each rule applies one or more formatting details to one or more HTML elements. Here's the structure of a simple rule:

```
selector {
  property: value;
  property: value;
}
```

And here's what each part means:

- The **selector** identifies the type of content you want to format. A browser hunts down all the elements in the web page that match your selector. There are many different ways to write a selector, but one of the simplest approaches (shown next) is to identify the elements you want to format by their element names. For example, you could write a selector that picks out all the level-one headings in your page.
- The **property** identifies the type of formatting you want to apply. Here's where you choose whether you want to change colors, fonts, alignment, or something else. You can have as many property settings as you want in a rule—this example has two.
- The **value** sets a value for the property. For example, if your property is color, the value could be light blue or a queasy green.

Now here's a real rule that does something:

```
h1 {
  text-align: center;
  color: green;
}
```

Pop this text into a style sheet and save it (for example, as *SampleStyles.css*). Then, take a sample web page (one that has at least one `<h1>` heading) and add a `<link>` element that refers to this style sheet. Finally, open this page in a browser. You'll see that the `<h1>` headings don't have their normal formatting—instead, they will be centered and green.

CSS Properties

The previous example introduces two formatting properties: `text-align` (which sets how text is positioned, horizontally) and `color` (which sets the text color).

There are many, many more formatting properties for you to play with. Table A-1 lists some of the most commonly used. In fact, this table lists almost all the style properties you'll encounter in the examples in this book (not including the newer CSS3 properties that are described in Chapters 6 and 7).

TABLE A-1 *Commonly used style sheet properties, by category*

	PROPERTIES
Colors	`color` `background-color`

Spacing	`margin`
	`padding`
	`margin-left`, `margin-right`, `margin-top`, `margin-bottom`
	`padding-left`, `padding-right`, `padding-top`, `padding-bottom`

Borders	`border-width`
	`border-style`
	`border-color`
	`border` (to set the width, style, and color in one step)

Text alignment	`text-align`
	`text-indent`
	`word-spacing`
	`letter-spacing`
	`line-height`
	`white-space`

Fonts	`font-family`
	`font-size`
	`font-weight`
	`font-style`
	`font-variant`
	`text-decoration`
	`@font-face` (for using fancy web fonts; see page 206)

Size	`width` `height`

Layout	`position`
	`left, right`
	`float, clear`

Graphics	`background-image`
	`background-repeat`
	`background-position`

TIP If you don't have a style sheet book on hand, you can get an at-a-glance overview of all the properties listed here (and more) at *www.htmldog.com/reference/cssproperties*. You can also get more information about each property, including a brief description of what it does and the values it allows.

Formatting the Right Elements with Classes

The previous style sheet rule formatted all the `<h1>` headings in a document. But in more complex documents, you need to pick out specific elements and give them distinct formatting.

To do this, you need to give these elements a name with the `class` attribute. Here's an example that creates a class named `ArticleTitle`:

```
<h1 class="ArticleTitle">HTML5 is Winning</h1>
```

Now you can write a style sheet rule that formats only this heading. The trick is to write a selector that starts with a period, followed by the class name, like this:

```
.ArticleTitle {
  font-family: Garamond, serif;
  font-size: 40px;
}
```

Now, the `<h1>` that represents the article title is sized up to be 40 pixels tall.

You can use the class attribute on as many elements as you want. In fact, that's the idea. A typical style sheet is filled with class rules, which take web page markup and neatly carve it into stylable units.

Finally, it's worth noting that you can create a selector that uses an element type and a class name, like this:

```
h1.ArticleTitle {
  font-size: 40px;
}
```

This selector matches any `<h1>` element that uses the `ArticleTitle` class. Sometimes, you may write this sort of style rule just to be clear. For example, you may decide to write your rule this way to make it clear that the `ArticleTitle` applies only to `<h1>` headings and shouldn't be used anywhere else. But most of the time, web designers just create straight classes with no element restrictions.

NOTE Different selectors can overlap. If more than one selector applies to the same element, they will both take effect, with the most general being applied first. For example, if you have a rule that applies to all headings and a rule that applies to the class named `ArticleTitle`, the all-headings rule is applied first, followed by the class rule. As a result, the class rule can override the properties that are set in the all-headings rule. If two rules are equally specific, the one that's defined last in the style sheet wins.

Style Sheet Comments

In a complicated style sheet, it's sometimes worth leaving little notes to remind yourself (or to let other people know) why a style sheet rule exists and what it's designed to do. Like HTML, CSS lets you add comments, which the web browser ignores. However, CSS comments don't look like HTML comments. They always start with the characters /* and end with the characters */. Here's an example of a somewhat pointless comment:

```
/* The heading of the main article on a page. */
.ArticleTitle {
  font-size: 40px;
}
```

Slightly More Advanced Style Sheets

You'll see an example of a practical style sheet in a moment. But first you need to consider a few of the finer points of style-sheet writing.

Structuring a Page with Elements

When working with style sheets, you'll often use the element to wrap up a section of content:

```
<div>
  <p>Here are two paragraphs of content.</p>
  <p>In a div container.</p>
</div>
```

On its own, the does nothing. But it gives you a convenient place to apply some class-based style sheet formatting. Here are some examples:

- **Inherited values.** Some CSS properties are *inherited*, which means the value you set in one element is automatically applied to all the elements inside. One example is the set of font properties—set them on a , and everything inside gets the same text formatting (unless you override it in places with more specific formatting rules).
- **Boxes.** A is a natural container. Add a border, some spacing, and a different background color (or image), and you have a way to make select content stand out.
- **Columns.** Professional websites often carve their content up into two or three columns. One way to make this happen is to wrap the content for each column in a , and then use CSS positioning properties to put them in their proper places.

TIP Now that HTML5 has introduced a new set of semantic elements, the element doesn't play quite as central a role. If you can replace a with another, more meaningful semantic element (like <header> or <figure>), you should do that. But when nothing else fits, the remains the go-to tool. Chapter 2 has a detailed description of all the semantic elements.

The element also has a smaller brother named . Like the element, the element has no built-in formatting. The difference is that is a block element, designed to wrap separate paragraphs or entire sections of content, while is an inline element that's meant to wrap smaller portions of content inside a block element. For example, you can use to apply custom formatting to a few words inside a paragraph.

NOTE CSS encourages good design. How? If you want to use CSS effectively, you need to properly plan your web page's structure. Thus, the need for CSS encourages even casual web-page writers to think seriously about how their content is organized.

Multiple Selectors

Sometimes, you might want to define some formatting that applies to more than one element or more than one class. The trick is to separate each selector with a comma.

For example, consider these two heading levels, which have different sizes but share the same title font:

```
h1 {
  font-family: Impact, Charcoal, sans-serif;
  font-size: 40px;
}

h2 {
  font-family: Impact, Charcoal, sans-serif;
  font-size: 20px;
}
```

You could pull the `font-family` setting into a separate rule that applies to both heading levels, like this:

```
h1, h2 {
  font-family: Impact, Charcoal, sans-serif;
}

h1 {
  font-size: 40px;
}

h2 {
  font-size: 20px;
}
```

It's important to understand that this isn't necessarily a better design. Often, it's better to duplicate settings because that gives you the most flexibility to change formatting later on. If you have too many shared properties, it's more awkward to modify one element type or class without affecting another.

Contextual Selectors

A contextual selector matches an element *inside* another element. Here's an example:

```
.Content h2 {
  color: #24486C;
  font-size: medium;
}
```

This selector looks for an element that uses the `Content` class. Then it looks for `<h2>` elements inside that element and formats them with a different text color and font size. Here's an example of an element it will format:

```
<div class="Content">
  ...
 <h2>Mayan Doomsday</h2>
  ...
</div>
```

In the first example, the first selector is a class selector, and the second selector (the contextual one) is an element type selector. However, you can change this up any way you want. Here's an example:

```
.Content .LeadIn {
  font-variant: small-caps;
}
```

This selector looks for an element in the `LeadIn` class, wrapped inside an element in the `Content` class. It matches this element:

```
<div class="Content">
  <p><span class="LeadIn">Right now</span>, you're probably feeling pretty
  good.  After all, life in the developed world is comfortable ...</p>
  ...
</div>
```

Once you get the hang of contextual selectors, you'll find that they're quite straightforward and ridiculously useful.

ID Selectors

Class selectors have a closely related cousin called *ID selectors*. Like a class selector, the ID selector lets you format just the elements you choose. And like a class selector, the ID selector lets you pick a descriptive name. But instead of using a period, you use a number-sign character (#):

```
#Menu {
  border-width: 2px;
  border-style: solid;
}
```

As with class rules, browsers don't apply ID rules unless you specifically tell them to in your HTML. However, instead of switching on the rules with a `class` attribute, you do so with the `id` attribute. For example, here's a `<div>` element that uses the Menu style:

```
<div id="Menu">...</div>
```

At this point, you're probably wondering why you would use an ID selector—after all, the ID selector seems almost exactly the same as a class selector. But there's one difference: You can assign a given ID to just *one* element in a page. In the current example, that means only one `<div>` can be labeled as a `Menu`. This restriction doesn't apply to class names, which you can reuse as many times as you like.

That means the ID selector is a good choice if you want to format a single, never-repeated element on your page. The advantage is that the ID selector clearly indicates the special importance of that element. For example, if a page has an ID selector named `Menu` or `NavigationBar`, the web designer knows there's only one menu or navigation bar on that page. Of course, you never *need* to use an ID selector. Some web designers use class selectors for everything, whether the section is unique or not. It's a matter of personal preference.

NOTE The `id` attribute also plays an important role in JavaScript, letting web page designers identify a specific element so it can be manipulated in code. The examples in this book use ID rules whenever an element already uses the `id` attribute for JavaScript, which avoids setting both the `id` attribute and the `class` attribute. In every other case, the examples use class rules, regardless of whether or not the element is unique.

Pseudo-Class Selectors

So far, the selectors you've seen have been straightforward. They've taken a single, obvious piece of information into consideration, like the element type, class name, or ID name. *Pseudo-classes* are a bit more sophisticated. They take extra information into account—information that might not be set in the markup or might be based on user actions.

For most of CSS history, browsers have supported just a few pseudo-classes, which were mostly designed for formatting links. The `:link` pseudo-class formats any link that points to a new, unvisited location. The `:visited` pseudo-class applies to any link that points to a location the reader has already visited. The `:hover` pseudo-class formats a link when a visitor moves the mouse over it, and the `:active` pseudo-class formats a link as a reader clicks it, before releasing the mouse button. As you can see, pseudo-classes always start with a colon (`:`).

Here's a style rule that uses pseudo-classes to create a misleading page—one where visited links are blue and unvisited links are red:

```
a:link {
  color: red;
}
a:visited {
  color: blue;
}
```

You can also use pseudo-classes with a class name:

```
.BackwardLink:link {
  color: red;
}
.BackwardLink:visited {
  color: blue;
}
```

Now an anchor element needs to specify the class name to display your new style, as shown here:

```
<a class="BackwardLink" href="...">...</a>
```

Pseudo-classes aren't just a way to format links. The `:hover` pseudo-class is useful for applying animated effects and creating fancy buttons. It's used with CSS3 transitions, as explained in Chapter 6 (page 195).

NOTE CSS3 also introduces some more advanced pseudo-classes that take other details into consideration, like the position of an element relative to other elements or the state of an input control in a web form. These pseudo-classes aren't described in this book, but you can learn about them from a Smashing Magazine article at *http://tinyurl.com/pc-css3*.

Attribute Selectors

Attribute selection is a feature offered by CSS3 that lets you format a specific type of element that also has a specific value set for one of its attributes. For example, consider the following style rule, which applies only to text boxes:

```
input[type="text"] {
  background-color:silver;
}
```

First, this selector grabs all the `<input>` elements. Then, it filters down its selection to include just those `<input>` elements that have a `type` attribute set to `"text"`, which it then formats. In the following markup, that means the first `<input>` element gets the silver background but the second doesn't:

```
<label for="name">Name:</label><input id="name" type="text"><br>
<input type="submit" value="OK">
```

Technically, you don't need to include the `type="text"` attribute in the first `<input>` element, because that's the default value. If you leave it out, the attribute selector still works, because it pays attention to the current value of the attribute and doesn't care how that value is defined in your markup.

Similarly, you could create a rule that formats the caption for this text box but ignores all other labels:

```
label[for="name"] {
  width: 200px;
}
```

NOTE You can still get a bit fancier with attribute selectors. For example, you can match a combination of attribute values, or match part of an attribute value. These techniques are awfully clever but inject too much complexity into the average style sheet. To get the lowdown, see the CSS3 standard for selectors at *http://tinyurl.com/s-css3*.

A Style Sheet Tour

Chapter 2 shows how you can learn to use HTML5's semantic elements by revising a straightforward, but nicely formatted page called *ApocalypsePage_Original.html* (Figure A-1). This page links to a style sheet named *ApocalypsePage_Original.css*:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Apocalypse Now</title>
  <link rel="stylesheet" href="ApocalypsePage_Original.css">
</head>
...
```

The style sheet is straightforward and relatively brief, weighing in somewhere over 50 lines. In this section, you'll dissect each one of its style rules.

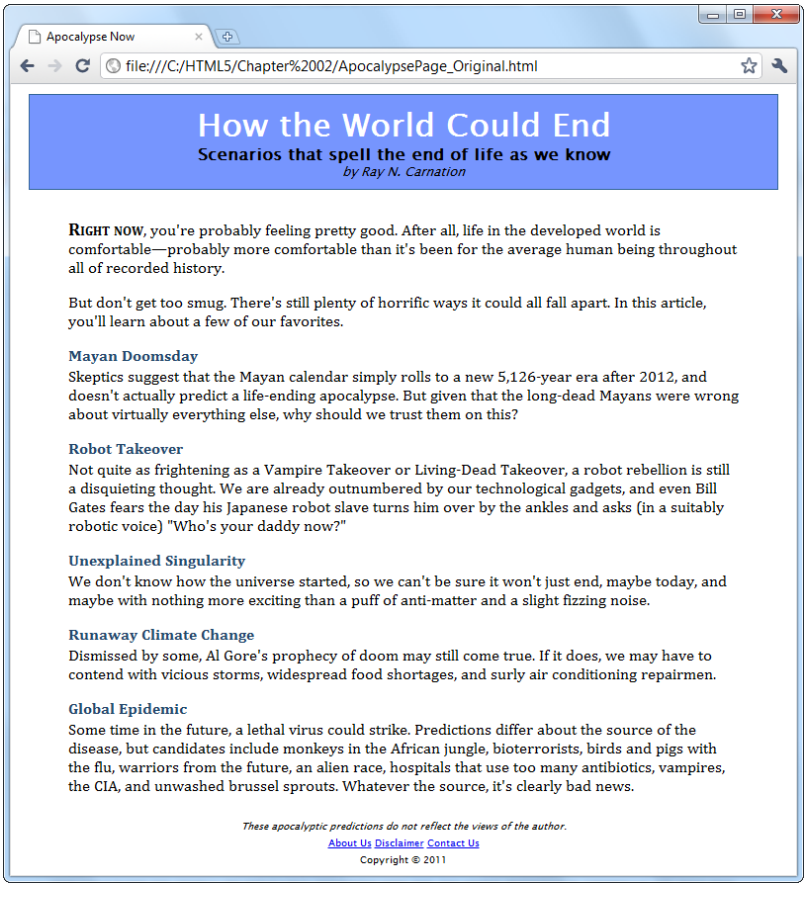


FIGURE A-1
The styles in this page are simple, but they follow the basic organizational principles that you'll see put into practice throughout this book.

First, the style sheet begins with a selector that targets the <body> element, which is the root of the entire web page. This is the best place to set inherited values that you want to apply, by default, to the rest of the document. Examples include margins, padding, background color, the font, and the width:

```
body {
  font-family: "Lucida Sans Unicode", "Lucida Grande", Geneva, sans-serif;
  max-width: 800px;
}
```

When setting the font-family property in CSS, you should follow two rules. First, use a web-safe font—one of the small number of fonts that are known to work on virtually all web-connected computers (see *http://tinyurl.com/ws-fonts* for a list). Second, use a font list that starts with the specific variant you want, followed by other possible fallbacks, and ends with serif or sans-serif (two font instructions that all browsers understand). If you prefer to use a fancy font that the user must download from your web server, check out the CSS3 web font feature on page 206.

The body rule also sets a maximum width, capping it at 800 pixels. This rule prevents overly long, unreadable lines when the browser window is made very wide. There are other possible techniques for handling this situation, including splitting the text into columns (page 217), using CSS media queries (page 231), or creating a sidebar to soak up the additional space. However, although setting a fixed 800-pixel width isn't the most glamorous solution, it's a common approach.

Next in the style sheet is a class-specific rule that formats the header region at the top of the page:

```
.Header {
  background-color: #7695FE;
  border: thin #336699 solid;
  padding: 10px;
  margin: 10px;
  text-align: center;
}
```

NOTE In this example, the header is simply a with the class name Header. However, Chapter 2 explains how you might consider replacing that with HTML5's <header> element.

There's a lot of information packed into this rule. The background-color property can be set, like all CSS colors, using a color name (which provides relatively few choices), an HTML color code (as done here), or the rgb() function (which specifies the red, green, and blue components of the color). The examples in this book use all three approaches, with color names in simple examples and color codes and the rgb() function in more realistic examples.

Incidentally, every HTML color code can be written with the `rgb()` function, and vice versa. For example, you can write the color in the above example using the `rgb()` function, like this:

```
background-color: rgb(118,149,254);
```

TIP To actually get the RGB values for the color you want, try an online color picker, or look the numbers up in your favorite drawing or graphics program.

The header rule also draws a thin border around its edges. It uses the all-in-one `border` property to specify the border thickness, border color, and border style (for example, `solid`, `dashed`, `dotted`, `double`, `groove`, `ridge`, `inset`, or `outset`) in one property setting.

With the background color and border details out of the way, the header rule sets 10 pixels of padding (between the content inside and the border) and 10 pixels of margin space (between the border and the surrounding web page). Finally, the text inside the header is centered.

The following three rules use contextual selectors to control how elements are formatted inside the header. The first one formats `<h1>` elements in the header:

```
.Header h1 {
  margin: 0px;
  color: white;
  font-size: xx-large;
}
```

TIP When setting a font size, you can use keywords (like the `xx-large` value used here). Or, if you want precise control, you can supply an exact measurement using pixels or em units.

The next two rules format two classes, named `Teaser` and `Byline`, which are also inside the header:

```
.Header .Teaser {
  margin: 0px;
  font-weight: bold;
}

.Header .Byline {
  font-style: italic;
  font-size: small;
  margin: 0px;
}
```

This code works because the header contains two elements. One has the class name Teaser, and contains the subtitle. The second has the author information, and uses the class name Byline. Here's the relevant portion of markup:

```
<div class="Header">
  <h1>How the World Could End</h1>
  <p class="Teaser">Scenarios that spell the end of life as we know it</p>
  <p class="Byline">by Ray N. Carnation</p>
</div>
```

Next up is a rule that formats a with the class name Content. It holds the main body of the page. The accompanying style sheet rule sets the font, padding, and line height:

```
.Content {
  font-size: medium;
  font-family: Cambria, Cochin, Georgia, "Times New Roman", Times, serif;
  padding-top: 20px;
  padding-right: 50px;
  padding-bottom: 5px;
  padding-left: 50px;
  line-height: 120%;
}
```

Whereas the header rule set the padding to be the same on all sides, the content rule sets different padding on each side, adding more space above and the most space on the sides. One way to do that is to specify the expanded padding properties (like padding-top, padding-right, and so on), as done here. Another option is to use the padding property with a series of values in a particular order—top, right, bottom, left. Here's how you can replace the expanded padding properties with just one property:

```
padding: 20px 50px 5px 50px;
```

Generally, you'll use this form when setting the padding on all sides, but you'll use the expanded padding properties if you want to change the padding on only certain sides. Of course, it's really a matter of taste.

The final line-height property sets the space between adjacent lines. The value of 120% gives some extra spacing, for a more readable feel.

Following the content rule are three contextual selectors that format elements inside. The first rule formats a span with the class name LeadIn. It's used to put the first two words in large, bold, small-cap lettering:

```
.Content .LeadIn {
  font-weight: bold;
  font-size: large;
  font-variant: small-caps;
}
```

The next two rules change how the <h2> and <p> elements are formatted in the content region:

```
.Content h2 {
  color: #24486C;
  margin-bottom: 2px;
  font-size: medium;
}

.Content p {
  margin-top: 0px;
}
```

As you can see, as a style sheet grows longer it doesn't necessarily become more complex. Here, the style sheet simply repeats the same basic techniques (class selectors and contextual selectors), but uses them to format other parts of the document.

Finally, the style sheet ends with the rules that format the footer portion. By now, you can interpret these on your own:

```
.Footer {
  text-align: center;
  font-size: x-small;
}

.Footer .Disclaimer {
  font-style: italic;
}

.Footer p {
  margin: 3px;
}
```

This rounds out the *ApocalypsePage_Original.css* style sheet. Feel free to download it from the try-out site (*http://prosetech.com/html5*) and try tweaking it to see what happens. Or, check out Chapter 2, which revises this page and the accompanying style sheet to use the HTML5 semantic elements.

APPENDIX

B

JavaScript: The Brains of Your Page

There was a time when the Web was all about markup. Pages held text and HTML tags, and not much more. Really advanced websites used server scripts that could tweak the HTML markup before it made its way to the browser, but the code stopped there.

Crack open a web page today and you're likely to find buckets of JavaScript code, powering everything from vital features to minor frills. Self-completing text boxes, pop-up menus, slideshows, real-time mapping, and webmail are just a few examples of the many ways crafty developers put JavaScript to work. In fact, it's nearly impossible to imagine a Web *without* JavaScript. While HTML is still the language of the Web, JavaScript is now the brains behind its most advanced pages.

In this appendix, you'll get a heavily condensed JavaScript crash course. This appendix won't provide a complete tutorial on JavaScript, nor does it have enough information to help you get started if you've never written a line of code in any programming language, ever. But if you have some rudimentary programming knowledge—say, you once learned a lick of Visual Basic, picked up the basics of Pascal, or took C out for spin—this appendix will help you transfer your skills to the JavaScript world. You'll get just enough information to identify familiar programming ingredients like variables, loops, and conditional logic. And you'll cover all the basic language elements that are used in the JavaScript-based examples in the rest of this book.

TIP If you need more help to get started with JavaScript, check out *JavaScript & jQuery: The Missing Manual* by David Sawyer McFarland, which also introduces jQuery, a popular JavaScript-enhancing toolkit. Or read Mozilla's detailed JavaScript guide at *http://developer.mozilla.org/JavaScript*.

How a Web Page Uses JavaScript

Before you can run a line of JavaScript, you need to know where to put it in your web page. It all starts with the <script> element. The following sections show you how to take a page from quick-and-dirty JavaScript injection to a properly structured example that you can put online without embarrassment.

Embedding Script in Your Markup

The simplest way to use the <script> element is to stick it somewhere in your HTML markup, like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Simple JavaScript Example</title>
</head>

<body>
  <p>At some point in the processing of this page, a script block
will run and show a message box.</p>

  <script>
alert("We interrupt this web page with a special JavaScript announcement.");
  </script>

  <p>If you get here, you've already seen it.</p>
</body>
</html>
```

This script block contains just one line of code, although you could just as easily pack it with a sequence of operations. In this case, the single line of code triggers JavaScript's built-in `alert()` function. The `alert()` function accepts a piece of text and shows that text in a message box (see Figure B-1). To move on, the user must click the OK button.

NOTE This example introduces a JavaScript convention that you'll see throughout this book, and on good websites everywhere: the semicolon. In JavaScript, semicolons indicate the end of each programming statement. Strictly speaking, semicolons aren't necessary (unless you want to cram multiple statements on a single line). However, they're considered good style.

If you want to run some JavaScript right away (as in this example), you'll probably put the <script> section at the end of the <body> section, just before the final </body> tag. That way, it runs only after the browser has processed all the page markup.

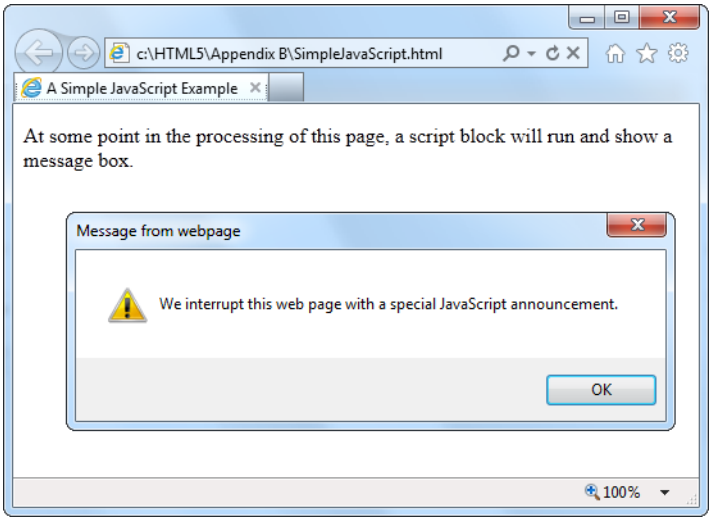


FIGURE B-1

When the web browser comes across JavaScript code, it runs it immediately. In fact, it even halts the page processing, temporarily. In this case, the code is held up until the web page user closes the message box by clicking OK. This allows the code to continue and the script block to end. The web browser then processes the rest of the markup.

GEM IN THE ROUGH

Dealing with Internet Explorer's Paranoia

If you run the alert example above in Firefox or Chrome, you'll find that everything works seamlessly. If you run it in Internet Explorer, you won't get the same satisfaction. Instead, you'll see a security warning in a yellow bar at the top or bottom of the page (depending on the version of IE). Until you go to that bar and click "Allow blocked content," your JavaScript code won't run.

At first glance, IE's security warning seems like a surefire way to scare off the bravest web visitor. But you don't need to worry; the message is just part of the quirky way Internet Explorer deals with web pages that you store on your hard drive. When you access the same page over the Web, Internet Explorer won't raise the slightest objection.

That said, the security warning is still an annoyance while you're testing your web page, because it forces you to keep explicitly telling the browser to allow the page to run JavaScript.

To avoid the security notice altogether, you can tell Internet Explorer to pretend you downloaded your page from a web server. You do this by adding a special comment called the *mark of the Web*. You place this comment in the <head> section of your page:

```
<head>
<meta charset="utf-8">
<!-- saved from url=(0014)about:internet
-->
  ...
</head>
```

When IE sees the mark of the Web, it treats the page as though it came from a web server, skipping the security warning and running your JavaScript code without hesitation. To all other browsers, the mark of the Web just looks like an ordinary HTML comment.

Using a Function

The problem with the previous example is that it encourages you to mingle code and markup in an unseemly mess. To keep things organized, you should wrap each code "task" in a *function*—a named unit of code that you can call into action whenever you need it.

When you create a function, you should give it a logical name. Here's a function named showMessage():

```
function showMessage() {
  // Code goes here ...
}
```

The function *body*—its guts—includes everything between the opening { bracket and the closing } bracket. Inside these delimiters, a function can hold as many lines of code as you need. Right now, the showMessage() function contains a single line, which is the "Code goes here" comment. (A JavaScript comment is a line that starts with two slash characters. The browser ignores all comments—you add them to remind yourself of important details or inform others about what the code is doing.)

To add some code to your function, just put all the statements you need between the curly brackets:

```
function showMessage() {
  alert("We interrupt this web page with a special JavaScript announcement.");
}
```

Of course, this whole shebang needs to go in a <script> block. The best place to put JavaScript functions is in the <head> section. This imposes some basic organization on your page, by moving the code out of the markup and into a separate section.

Here's a revamped version of the earlier example, which now uses a function:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Simple JavaScript Example</title>
 <script>
function showMessage() {
  alert("We interrupt this web page with a special JavaScript announcement.");
}
 </script>
</head>
...
```

Functions, on their own, don't do anything. To trigger a function, you need another piece of code that *calls* the function.

Calling a function is easy—in fact, you've already seen how to do it with the built-in alert() function. You simply write the function name, followed by a set of

parentheses. Inside the parentheses, you put whatever data the function needs. Or, if the function doesn't accept any data, like `showMessage()`, you simply include parentheses with nothing inside them:

```
...
<body>
  <p>At some point in the processing of this page, a script block
will run and show a message box.</p>
 <script>
showMessage();
 </script>
  <p>If you get here, you've already seen it.</p>
</body>
</html>
```

In this example, the function and the code that calls the function are in separate `<script>` blocks. This design isn't necessary, but it's used here to emphasize the separation between these two pieces.

At first glance, adding a function seems to make this example more complicated than before. But it's actually a dramatic step forward in design, for several reasons:

- **The bulk of the code is out of the markup.** You need just one line of code to call a function. However, a realistic function will contain a pile of code, and a realistic page will contain a pile of functions. You definitely want to separate all those details from your markup.
- **You can reuse your code.** Once code is in a function, you can call that function at different times, from different places in your code. This isn't obvious in this simple example, but it becomes an important factor in more complex applications, like the painting application in Chapter 8.
- **You're ready to use external script files.** Moving your code out of the markup is a precursor to moving it right out of the HTML file, as you'll see in the next section, for even better organization.
- **You can add events.** An event is a way for you to tell the page to run a specific function when a specific occurrence takes place. Web pages are event-driven, which means most code is fired up when an event happens (rather than being launched through a script block). Events pair neatly with functions, as you'll see on page 457.

Moving the Code to a Script File

Pulling your JavaScript code together into a set of functions is the first step in good organization. The second step is to take that script code and put it in an entirely separate file. Do this with all your scripts, and you'll get smaller, simpler web pages—and the ability to reuse the same functions in different web pages. In fact, putting script code in an external file is analogous to putting CSS style rules in an external file. In both cases, you gain the ability to reuse your work and you leave simpler pages behind.

NOTE Virtually every well-designed web page that uses JavaScript puts the code in one or more script files. The only exceptions are if you have a few lines of very simple code that you're certain not to use anywhere else, or if you're creating a one-off example.

Script files are always plain text files. Usually, they have the extension *.js* (which stands for JavaScript). You put all your code inside a script file, but you don't include the `<script>` element. For example, here are the complete contents of a script file named *MessageScripts.js*:

```
function showMessage() {
  alert("We interrupt this web page with a special JavaScript announcement.");
}
```

Now save the file, and put it in the same folder as your web page. In your web page, define a script block, but don't supply any code. Instead, add the `src` attribute and indicate the script file you want to link to:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>A Simple JavaScript Example</title>
  <script src="MessageScripts.js"></script>
</head>

<body>
  <p>At some point in the processing of this page, a script block
will run and show a message box.</p>
  <script>
showMessage()
  </script>
  <p>If you get here, you've already seen it.</p>
</body>
</html>
```

When a browser comes across this script block, it requests the *MessageScripts.js* file and treats it as though the code were right inside the page. That means you can call the `showMessage()` function in exactly the same way you did before.

NOTE Even though the script block doesn't actually contain any code when you use external script files, you must still include the closing `</script>` tag. If you leave that out, the browser assumes everything that follows—the rest of the page—is part of your JavaScript code.

You can also link to JavaScript functions on another website—just remember that the `src` attribute in the `<script>` element needs to point to a full URL (like *http://SuperScriptSite.com/MessageScript.js*) instead of just a file name. This technique is necessary for plugging into other companies' web services, like Google Maps (page 410).

Responding to Events

So far, you've seen how to run script right away—by weaving a script block into your HTML markup. But it's far more common to trigger code after the page is finished processing, when the user takes a specific action—like clicking a button or moving the mouse pointer over an element.

To do so, you need to use JavaScript *events*, which are notifications that an HTML element sends out when specific things happen. For example, JavaScript gives every element an event named `onMouseOver` (a compressed version of "on mouse over"). As the name suggests, this event takes place (or *fires*, to use programmer-speak) when a visitor moves his mouse pointer over an HTML element like a paragraph, link, image, table cell, or text box. That action triggers the `onMouseOver` event and your code flies into action.

This discussion brings up one key question: How do you link your code to the event you want to use? The trick is to add an event attribute to the appropriate element. So if you want to handle the `onMouseOver` event of an `<img>` element, you need markup like this:

```
<img src="sunny.jpg" alt="A sunny day" onmouseover="showMessage()">
```

NOTE In JavaScript, function, variable, and object names are case-sensitive, meaning `showMessage` is not the same as `showMESSAGE` (and the latter fails). However, the event attribute names are not case sensitive, because they are technically a part of HTML markup, and HTML tolerates any combination of uppercase and lowercase letters. Even so, it's common to write event attributes with no capitals (as shown here) because this matches the old rules of XHTML, and most programmers are too lazy to reach for the Shift key anyway.

Now, when the mouse moves over the image, and the `onMouseOver` event fires, the browser automatically calls the `showMessage()` function. This function pops up a rather unremarkable message box (Figure B-2). When an event triggers a function in this way, that function is called an *event handler*.

To use events effectively, you need to know which events JavaScript supports. In addition, you need to know which events work on which HTML elements. Table B-1 provides a list of commonly used events and the HTML elements that they apply to. (You can find a more complete reference at *http://developer.mozilla.org/DOM/element*.)

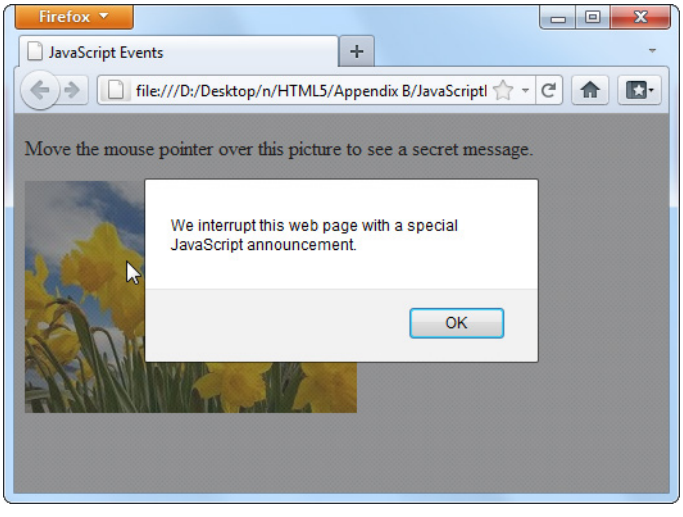


FIGURE B-2

In this example, the alert box doesn't pop up until you move your mouse pointer over the link.

TABLE B-1 *Common HTML object events*

EVENT NAME	DESCRIPTION	APPLIES TO
onClick	Triggered when you click an element.	Virtually all elements
onMouseOver	Triggered when you move your mouse pointer over an element.	Virtually all elements
onMouseOut	Triggered when you move your mouse pointer away from an element.	Virtually all elements
onKeyDown	Triggered when you press a key.	<select>, <input>, <textarea>, <a>, <button>
onKeyUp	Triggered when you release a pressed key.	<select>, <input>, <textarea>, <a>, <button>
onFocus	Triggered when a control receives *focus* (in other words, when you position the cursor on the control so you can type something in). Controls include text boxes, checkboxes, and so on—see page 108 to learn more.	<select>, <input>, <textarea>, <a>, <button>
onBlur	Triggered when focus leaves a control.	<select>, <input>, <textarea>, <a>, <button>

EVENT NAME	DESCRIPTION	APPLIES TO
onChange	Triggered when you change a value in an input control. In a text box, this event doesn't fire until you move to another control.	<select>, <input type="text">, <textarea>
onSelect	Triggered when you select a portion of text in an input control.	<input type="text">, <textarea>
onError	Triggered when the browser fails to download an image (usually due to an incorrect URL).	<img>
onLoad	Triggered when the browser finishes downloading a new page or finishes loading an object, like an image.	<img>, <body>
onUnload	Triggered when a browser unloads a page. (This typically happens after you enter a new URL or when you click a link. It fires just *before* the browser downloads the new page.)	<body>

A Few Language Essentials

A brief appendix isn't enough to cover any language, even one as straightforward as JavaScript. However, the following sections will fill you in on a few language essentials that you'll need to digest the examples elsewhere in this book.

Variables

Every programming language has the concept of *variables*—containers that you can use to store bits of information in memory. In JavaScript, every variable is created the same way, by declaring it with the `var` keyword followed by the variable name. This example creates a variable named `myMessage`:

```
var myMessage;
```

NOTE JavaScript variables are case-sensitive, which means a variable named `myMessage` differs from one named `MyMessage`. If you try to use them interchangeably, you'll wind up with an error message (if your browser is nice) or a bizarre mistake in the page (which is usually what happens).

To store information in a variable, you use the equal sign (`=`), which copies the data on the right side of the equal sign into the variable on the left. Here's a one-step example that defines a variable and puts a text value (which is known as a *string*) inside:

```
var myMessage = "Everybody loves variables";
```

You can then use your variable:

```
// Show the variable text in a message box.
alert(myMessage);
```

NOTE JavaScript is a notoriously loose language, and it lets you use variables even if you don't specifically declare them with the `var` keyword. However, doing so is considered extremely bad form and is likely to lead to sloppy mistakes.

Null Values

One special value you may run into is `null`, which is programmer-speak for "nothing." If a variable is null, it indicates that a given object doesn't exist or is empty. Depending on the context, this may signal that a specific feature is unavailable. For example, Modernizr (page 31) uses null value tests to determine whether the browser supports certain HTML5 features. You may also check for null values in your scripts—for example, to determine whether you haven't yet created or stored an object:

```
if (myObject == null) {
  // There is no myObject in existence.
  // Now might be a good time to create one.
}
```

Variable Scope

There are two basic places you can create a variable—inside or outside a function. The following code snippet has one of both:

```
<script>
var outsideVariable;

function doSomething() {
  var insideVariable;
  ...
}
</script>
```

If you create a variable inside a function (called a *local variable*), that variable exists only while that function is running. Here, `insideVariable` is a local variable. As soon as the `doSomething()` method ends, the variable is tossed out of memory. That means the next time the page calls `doSomething()`, the `insideVariable` is created from scratch, with none of its previous data.

On the other hand, if you create a variable outside a function (called a *global variable*), its value lasts as long as the page is loaded in the browser. Furthermore, every function can use that variable. In the previous example, `outsideVariable` is a global variable.

TIP The rule of thumb is to use a local variable, unless you specifically need to share your variable with multiple functions, or to retain its value after the function ends. That's because it's more trouble to keep track of global variables, and if you use too many, your code becomes messy.

Variable Data Types

In JavaScript, variables can store different data types, such as text, integers, floating point numbers, arrays, and objects. However, no matter what you want to store in your variable, you define it with the same `var` keyword. You do *not* set the data type of your variable.

That means you can take the `myMessage` variable, with its piece of text, and replace that with a numeric value, like this:

```
myMessage = 27.3;
```

This behavior makes JavaScript easy to use, because any variable can hold any type of content. It can also let JavaScript mistakes slip past undetected. For example, you might want to grab the text out of a text box and put that in a variable, like this:

```
var = inputElement.value;
```

But if you're not careful, you can accidentally end up putting the entire text box *object* into the variable, like this:

```
var = inputElement;
```

JavaScript allows both actions, so it won't complain. But a few lines into your code, this mistake will probably lead to some sort of unrecoverable problem. At that point, the browser simply stops running the rest of your code, without giving you any error message to explain what happened. In cases like these, you need the help of a Java-Script debugging tool (see the box on page 462), which can pause your code at any time and let you peer into your variables, so you can see what data they currently contain.

Operations

One of the most useful things you can do with numeric variables is perform *operations* on them to change your data. For example, you can use arithmetic operators to perform mathematical calculations:

```
var myNumber = (10 + 5) * 2 / 5;
```

These calculations follow the standard order of operations (parentheses first, then multiplication and division, then addition and subtraction). The result of this calculation is 6.

You can also use operations to join together multiple pieces of text into one long string. In this case, you use the plus (+) operator:

```
var firstName = "Sarah";
var lastName = "Smithers";
var fullName = firstName + " " + lastName;
```

TROUBLESHOOTING MOMENT

Identifying Errors in JavaScript Code

In order to deal with problems (like the variable mistake shown on page 461), you need to master *debugging*—the fine art of hunting down the problems in your code and stamping them out. Unfortunately, the way you go about debugging depends on the browser you're using. Different browsers have different debugging tools (or support different debugging extensions). And while they all serve the same purpose, they don't work in exactly the same way.

Fortunately, all the information you need is on the Web. Here are some links that can explain how to debug JavaScript mistakes, based on your browser of choice:

- **Internet Explorer.** To sort out problems with IE, press F12 to pop up the Developer Tools window. To learn how to use it, visit *http://tinyurl.com/debug-ie*.
- **Firefox.** Serious Firefox developers use a Firefox add-in called Firebug to see what their code is doing at all times. Get it (and learn more) at *http://getfirebug.com/javascript*.
- **Google Chrome.** Chrome has a respectable built-in debugger. To get started, read Google's debugging tutorial at *http://tinyurl.com/c-debugger*.
- **Opera.** Opera's debugging tool of choice is Dragonfly. You can learn about it at *www.opera.com/dragonfly*.
- **Safari.** Safari has a powerful set of built-in debugging tools, although tracking down the documentation that explains them can be tricky. You can start with a fairly technical article from the Safari Developer Library at *http://tinyurl.com/safari-debug*.

Remember, it doesn't matter what browser and debugging tool you use to correct problems. Once they're fixed in one browser, they're fixed for everyone.

Now the `fullName` variable holds the text "Sarah Smithers." (The `" "` in the code above tells JavaScript to leave a space between the two names).

When making simple modifications to a variable, there's a shortcut you're likely to use. For example, if you have this basic addition operation:

```
var myNumber = 20;
myNumber = myNumber + 10;
// (Now myNumber is 30.)
```

You can rewrite it like this:

```
var myNumber = 20;
myNumber += 10;
// (Now myNumber is 30.)
```

This trick of moving the operator to the left side of the equal sign works with pretty much any operator. Here are some examples:

```
var myNumber = 20;
myNumber -= 10;
// (Now myNumber is 10.)
myNumber *= 10;
// (Now myNumber is 100.)
```

```
var myText = "Hello";
var myText += " there.";
// (Now myText is "Hello there.")
```

And if you want to add or subtract the number 1, there's an even more concise shortcut:

```
var myNumber = 20;
myNumber++;
// (Now myNumber is 21.)

myNumber--;
// (Now myNumber is 20.)
```

Conditional Logic

All conditional logic starts with a *condition*: an expression that is either true or false. Based on the result, you can decide to run some code or to skip over it.

To create a condition, you need to rely on JavaScript's *logical operators*, which are detailed in Table B-2.

TABLE B-2 *Logical operators*

OPERATOR	DESCRIPTION
==	Equal to.
!=	Not equal to.
===	Exactly equal to (in value *and* data type).
!==	Not exactly equal to.
!	Not. (This reverses the condition, so if it would ordinarily be true, it is now false, and vice versa.)
<	Less than.
>	Greater than.
<=	Less than or equal to.
>=	Greater than or equal to.
&&	Logical and (evaluates to true only if both expressions are true). If the first expression is false, the second expression is not evaluated.
||	Logical or (evaluates to true if either expression is true). If the first expression is true, the second expression is not evaluated.

Here's an example of a simple condition:

```
myNumber < 100
```

To use this condition to make decisions, you need to put it with an `if` statement. Here's an example:

```
if (myNumber < 100) {
  // (This code runs if myNumber is 20, but not if it's 147.)
}
```

NOTE Technically, you don't need the curly brackets around your conditional code, unless you have more than one statement. However, including the brackets is always clearer and avoids potential errors if you do have multiple statements.

When testing equality, make sure you use two equal signs. A single equal sign sets a variable's value, rather than performing the comparison you want:

```
// Right:
if (myName == "Joe") {
}

// Wrong:
if (myName = "Sarah") {
}
```

Although two equal signs are good, it turns out that *three* may be even better. Many JavaScript pros prefer to test equality using the "exactly equal to" operator (that's `===`) rather than the mere "equal to" operator (`==`). The difference is that the "equal to" operator will convert data types to try to make a match, while the more stringent "exactly equal to" operator insists on a perfect match of value and data type.

Here's an example that illustrates the difference:

```
var myNumberAsText = "45";

// This is true, because the "equal to" operator is willing to convert
// "45" the string to 45 the number.
if (myNumberAsText == 45) {
}

// This is false, because the data types don't match.
if (myNumberAsText === 45) {
}
```

In most cases, it doesn't matter whether you use the "equal to" or "exactly equal to" operator, but there are some rare type conversion mistakes that "exactly equal to" can prevent. For that reason, JavaScript experts generally prefer using three equal signs instead of two.

If you want to evaluate more than one condition, one after the other, you can use more than one `if` block (naturally). But if you want to look at a series of conditions

and find the first one that matches (while ignoring the others), you need the `else` keyword. Here it is at work:

```
if (myNumber < 100) {
  // (This code runs if myNumber is less than 100.)
}
else if (myNumber < 200) {
  // (This code runs if myNumber is less than 200 but greater than or equal to
  // 100.)
}
else {
  // (In all other cases, meaning myNumber is 200 or more, this code runs.)
}
```

You can include as many or as few conditions as you like in an `if` block, and adding a final `else` without a condition is also optional.

Loops

A loop is a basic programming tool that lets you repeat a block of code. The king of JavaScript loops is the `for` loop. It's essentially a loop with a built-in counter. Most programming languages have their own version of this construct.

When creating a `for` loop, you set the starting value, the ending value, and the amount to increment the counter after each pass. Here's one example:

```
for (var i = 0; i < 5; i++){
  // (This code executes five times.)
  alert("This is message: " + i);
}
```

At the beginning of the `for` loop is a set of brackets with three important pieces of information. The first portion (in this example, `var i = 0`) creates the counter variable (`i`) and sets its initial value (0). The second portion (`i < 5`) sets a termination condition. If it's not true (for example, `i` is increased to 5), the loop ends and the code inside is not repeated again. The third portion (`i++`), increments the counter variable. In this example, the counter is incremented by 1 after each pass. That means `i` will be 0 for the first pass, 1 for the second pass, and so on. The end result is that the code runs five times and shows this series of messages:

```
This is message: 0
This is message: 1
This is message: 2
This is message: 3
This is message: 4
```

Arrays

The `for` loop pairs naturally with the *array*—a programming object that stores a list of values.

JavaScript arrays are remarkably flexible. Unlike in many other programming languages, you don't define the number of items you want an array to store in JavaScript. Instead, you simply begin by creating an empty array with square brackets, like this:

```
var colorList = [];
```

You can then add items using the array's `push()` method:

```
colorList.push("blue");
colorList.push("green");
colorList.push("red");
```

Or you can place an array item in a specific position. If this memory slot doesn't already exist, JavaScript creates if for you, happily:

```
colorList[3] = "magenta";
```

And you can pull it out yourself, also by position:

```
var color = colorList[3];
```

NOTE Just remember that JavaScript arrays use zero-based counting: The first item in an array is in slot 0, the second is in slot 1, and so on.

Once you have an array stocked with items, you can process each of them using a `for` loop like this:

```
for (var i = 0; i < colorList.length; i++) {
  alert("Found color: " + colorList[i]);
}
```

This code moves from the first item (the item at position 0) to the last item (using the array's length property, which reports its total item count). It shows each item in a message box, although you could surely think of something more practical to do with your array items.

Using a `for` loop to process an array is a basic technique in JavaScript. You'll use it often in this book, with arrays that you create yourself and ones that are provided to you by other JavaScript functions.

Functions That Receive and Return Data

Earlier, you saw a simple function, `showMessage()`. When you called `showMessage()`, you didn't need to supply any data, and when it finished, it didn't provide you with any additional information.

Functions aren't always that simple. In many cases, you need to send specific information to a function, or take the results of a function and use them in another operation. For example, imagine you want to create a version of the `showMessage()` function that you can use to show different messages. To do so, you need to make the `showMessage()` function accept a single *parameter*. This parameter represents the customized text you want to incorporate into your greeting.

To add the parameter, you must first give it a name, say customMessage, and put it in parentheses after the function name, like so:

```
function showMessage(customMessage) {
  alert(customMessage);
}
```

NOTE There's no limit to how many pieces of information a function can accept. You just need to separate each parameter with a comma.

Inside the function, it can work with the parameters just like normal variables. In this example, the function simply takes the supplied text and shows it in a message box.

Now, when calling the showMessage( ) function, you need to supply one value (called an *argument*) for each of the function's parameters:

```
showMessage("Nobody likes an argument.");
```

Parameters let you send information *to* a function. You can also create functions that send information *back* to the script code that called them. The key to doing this is the return command, which you put right at the end of your function. The return command ends the function immediately, and spits out whatever information your function generates.

Of course, a sophisticated function can accept *and* return information. For example, here's a function that multiplies two numbers (the numberA and numberB parameters) and returns the result to anyone who's interested:

```
function multiplyNumbers(numberA, numberB) {
  return numberA * numberB;
}
```

Here's how you use this function elsewhere on your web page:

```
// Pass in two numbers, and get the result.
var result = multiplyNumbers(3202, 23405);

// Use the result to create a message.
var message = "The product of 3202 and 23405 is " + result;

// Show the message.
showMessage(message);
```

Of course you don't really need a function to multiply numbers (an ordinary JavaScript calculation can do that), nor do you need a function to show a message box (because the built-in alert( ) function can handle that job). But both examples do a good job of showing you how functions tick, and you'll use parameters and return values in the same way in more complex functions.

Objects

Virtually all modern programming languages include the concept of an *object*, which is a package of related data and features that you can interact with in code. For example, every HTML element on a web page is an object in the eyes of your code. Using the different properties of this object, you can read or alter its content, change its style, and handle its events.

Programmers often need to create their own objects, too. For example, you'll create circle objects in the circle-drawing example in Chapter 9 (page 294) and ball objects for the bouncing ball example on page 304.

Objects make complex programming tasks easier, particularly when you need to manage multiple copies of the same data structure. For example, if you need to fill a page with bouncing balls, it would be a serious headache to create dozens and dozens of variables to hold the position and speed of each individual ball. But if you have a way to declare a *template* for your balls, you can reuse that single template to create as many live ball objects as you need, whether that's just one or eighty-four thousand.

Most languages have a specific syntax for creating object templates. Often, these templates are called *classes*. But JavaScript doesn't include an official class feature. This oversight is due to JavaScript's history—it started life as a simple, streamlined scripting language, not a serious tool for building online apps. Fortunately, clever JavaScript programmers have found ways to fill the object gaps. Although their tricks began as inventive-but-odd hacks, they're now considered standard practice.

For example, if you need to define an object in JavaScript, you write an object-definition function for it. This object-definition function is the template that takes the place of a true class in languages like C#, Java, and Visual Basic. Here's an object-definition function that lets you create `Person` objects:

```
function Person() {
  this.firstName = "Joe";
  this.lastName = "Grapta";
}
```

The object-definition function has a single task. It defines, one at a time, all the individual bits of data that constitute that object. In the case of the `Person` object shown above, this includes two details: a first and last name. (You could easily add additional details, like a date of birth, email address, and so on.) The `this` keyword is the magic touch—it makes sure that each property you create will become part of the object.

As long as you start the line with `this` followed by a dot, you can name the property variable whatever you want. So the following example is an equally valid person that stores the same data, but with different property names:

```
function Person() {
  this.F_name = "Joe";
```

```
  this.L_name = "Grapta";
}
```

Now you can use the `Person()` function to create a new person object. The trick is that you don't want to call the function and trigger its code. Instead, you want to create a new copy of the function by using the `new` keyword. Here's how that works:

```
// Create a new Person, and store it in a variable named joePerson.
var joePerson = new Person();
```

Once you have a live object, you can access all its details through the property names that you used in the object-definition function:

```
// Read the firstName property.
alert("His name is " + joePerson.firstName);

// Change the firstName property.
joePerson.FirstName = "Joseph";
```

You can improve your object-definition function so your code specifies some or all the data details through arguments. This saves you the trouble of creating your object and then customizing it with additional lines of code. It also makes sure your object starts out in the correct state, avoiding potential mistakes. Here's an example that updates the `Person()` function in this way:

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

And here's how you use the new `Person()` function to create two objects:

```
var newCustomer1 = new Person("Christy", "Shanks");
var newCustomer2 = new Person("Emilio", "Taginelle");
```

Page 294 has a full walkthrough of an example that uses basic object creation, and you'll see the same technique at work throughout this book.

Object Literals

In the previous section, you saw how to create objects in JavaScript using a function, which acts as a template. When you want to formally define the ingredients that make up an object, using a function is the best approach. It leads to well-organized code and makes complex coding tasks easier. It's the best choice when you want to work with your objects in different ways and in different places in your code. But sometimes you just need a quick way to create an object for a one-off task. In this case, an *object literal* makes sense, because it requires nothing more advanced than a pair of curly braces.

To create an object literal, you use an opening curly brace, supply a comma-separated list of properties, and then end with a closing curly brace. You can use spacing and line breaks to make your code more readable, but that's not required. Here's an example:

```
var personObject = {
  firstName="Joe",
  lastName="Grapta"
};
```

For each property, you specify the property name and its starting value. Thus, the above code sets `personObject.firstName` to the text "Joe" and `personObject.lastName` to "Grapta."

The example on page 408 uses object literals to send information to the geolocation system. As long as you use the right property names (the ones the `getCurrentPosition()` method is expecting), an object literal works perfectly.

TIP If you want to learn more about object literals, object functions, and everything else to do with custom objects in JavaScript, check out the detailed information at *www.javascriptkit.com/javatutors/oopjs.shtml*.

Interacting with the Page

So far, you've seen the right way to put JavaScript in a page, but you haven't done anything impressive (in fact, you haven't done anything but pop up a message box). Before going ahead, you need to know a bit more about the role JavaScript typically plays.

First, it's important to understand that JavaScript code is *sandboxed*, which means its capabilities are carefully limited. Your page can't perform any potentially risky tasks on your visitor's computer, like sending orders to a printer, accessing files, running other programs, reformatting a hard drive, and so on. This design ensures good security, even for careless visitors.

Instead, JavaScript spends most of its time doing one of these tasks:

- **Updating the page.** Your script code can change elements, remove them, or add new ones. In fact, JavaScript has complete flexibility to change every detail about the currently displayed HTML, and can even replace the whole document.
- **Retrieving data from the server.** JavaScript can make new web requests from the same web server that sent the original page. By combining this technique with the one above, you can create web pages that seamlessly update important information, like a list of news stories or a stock quote.
- **Sending data to the server.** HTML already has a way to send data to a web server, called web forms (Chapter 4). But JavaScript can take a much more subtle approach. It can grab bits of information out of your form controls, validate them, and even transmit them to the web server, all without forcing the browser to refresh the page.

The last two techniques require the XMLHttpRequest object, a JavaScript extension that's described on page 377. In the following sections, you'll take a look at the first of these, which is a fundamental part of almost every JavaScript-powered page.

Manipulating an Element

In the eyes of JavaScript, your page is much more than a static block of HTML. Instead, each element is a live object that you can examine and modify with JavaScript code.

The easiest way to get hold of an object is to identify it with a unique name, which you apply through the id attribute. Here's an example:

```
<h1 id="pageTitle">Welcome to My Page</h1>
```

Once you give your element a unique ID, you can easily locate that object in your code and have JavaScript act on it.

JavaScript includes a handy trick for locating an object: the document.getElementById() method. Basically, document is an object that represents your whole HTML document. It's always available, and you can use it anytime you want. This document object, like any object worthy of the name, gives you some handy properties and methods. The getElementById() method is one of the coolest—it scans a page looking for a specific HTML element.

NOTE If you're familiar with the basics of object-oriented programming, properties and methods are old hat. If not, you can think of *properties* as data attached to an object, and you can think of *methods* as functions built into an object.

When you call the document.getElementById() method, you supply the ID of the HTML element you're looking for. Here's an example that digs up the object for an HTML element with the ID pageTitle:

```
var titleObject = document.getElementById("pageTitle");
```

This code gets the object for the <h1> element shown earlier and stores it in a variable named titleObject. By storing the object in a variable, you can perform a series of operations on it without having to look it up more than once.

So what, exactly, can you do with HTML objects? To a certain extent, the answer depends on the type of element you're working with. For example, if you have a hyperlink, you can change its URL. If you have an image, you can change its source. And there are some actions you can take with almost all HTML elements, like changing their style or modifying the text that appears between the beginning and ending tags. As you'll see, you'll find these tricks useful in making your pages more dynamic—for example, you can change a page when a visitor takes an action, like clicking a link. Interactions like these make visitors feel as though they're using an intelligent, responsive program instead of a plain, inert web page.

Here's how you modify the text inside the just-mentioned <h1> element, for example:

```
titleObject.innerHTML = "This Page Is Dynamic";
```

This script works because it uses the *property* named innerHTML, which sets the content that's nested inside an element (in this case, an <h1> element with the page title). Like all properties, innerHTML is just one aspect of an HTML object you can alter. To write code statements like this, you need to know what properties JavaScript lets you play with.

Obviously, some properties apply to specific HTML elements only, like the src attribute that's used to load a new picture into this <img> element:

```
var imgObject = document.getElementById("dayImage");
dayImage.src = "cloudy.jpg";
```

You can also tweak CSS properties through the style object:

```
titleObject.style.color = "rgb(0,191,255)";
```

Modern browsers boast a huge catalog of DOM properties you can use with just about any HTML element. Table B-3 lists some of the most useful.

TABLE B-3 *Common HTML object properties*

PROPERTY	DESCRIPTION
className	Lets you retrieve or set the class attribute (see page 438). In other words, this property determines what style (if any) this element uses. Of course, you need to define this style in an embedded or linked style sheet, or you'll end up with the plain-Jane default formatting.
innerHTML	Lets you read or change the HTML inside an element. The innerHTML property is insanely useful, but it has two quirks. First, you can use it on all HTML content, including text and tags. So if you want to put bold text inside a paragraph, you can set innerHTML to Hi. Second, when you set innerHTML, you replace all the content inside this element, including any other HTML elements. So if you set the innerHTML of a element that contains several paragraphs and images, all of these items disappear, to be replaced by your new content.
parentElement	Provides the HTML object for the element that contains this element. For example, if the current element is a element in a paragraph, this gets the object for the <p> element. Once you have this element, you can modify it too.
style	Bundles together all the CSS attributes that determine the appearance of the HTML element. Technically, the style property returns a full-fledged style object, and you need to add another dot (.) and the name of the style attribute you want to change, as in myElement.style.fontSize. You can use the style object to dictate colors, borders, fonts, and even positioning.
tagName	Provides the name of the HTML element for this object, without the angle brackets. For example, if the current object represents an <img> element, this returns the text "img."

TIP HTML elements also provide a smaller set of useful methods, including some for modifying attributes, like `getAttribute()` and `setAttribute()`; and some for adding or removing elements, like `insertChild()`, `appendChild()`, and `removeChild()`. To learn more about the properties and methods that a specific HTML element supports, check out the reference at *http://developer.mozilla.org/DOM/element*.

Connecting to an Event Dynamically

On page 457, you saw how to wire up a function using an event attribute. However, it's also possible to connect an event to a function using JavaScript code.

Most of the time, you'll probably stick to event attributes. However, there are cases where that isn't possible or convenient. One of the most common examples is when you create an HTML object in your code and then add it to the page dynamically. In this situation, there's no markup for the new element, so there's no way to use an event attribute. (You'll see this technique in the canvas-drawing example in Chapter 8.) Another case is when you're attaching an event to a built-in object rather than an element. (You'll see this example when you handle storage events in Chapter 10.) For all these reasons, it's important to understand how to wire up events with code.

NOTE There are several different ways to attach events, but they aren't all supported by all browsers. This section uses the *event property* approach, which is supported by all. Incidentally, if you decide to use a JavaScript toolkit like jQuery, you'll probably find that it adds yet another event-attaching system, which will work on all browsers and may provide a few extra features.

Fortunately, attaching events is easy. You simply set an event property that has the same name as the event attribute you would normally use. For example, say you have an `<img>` element like this somewhere on your page:

```
<img id="dayImage" src="sunny.jpg" alt="The weather">
```

Here's how you tell the browser to call the `swapImage()` method when that image is clicked:

```
var imgObject = document.getElementById("dayImage");
imgObject.onclick = swapImage;
```

However, don't make this mistake:

```
imgObject.onclick = swapImage();
```

This runs the `swapImage()` function, takes the result (if it returns one), and uses that to set the event handler. This is almost certainly not what you want.

To understand what really happens when the `<img>` element is clicked, you need to look at the code in the `swapImage()` function. It grabs the `<img>` element and modifies the `src` attribute to point to a new picture (see Figure B-3):

```
// Keep track of whether the picture has been swapped from day to night.
var dayTime = true;
```

```
// This function runs when the onClick event happens.
function swapImage() {
  var imgObject = document.getElementById("dayImage");

  // Flip from day to night or night to day, and update the picture to match.
  if (dayTime === true) {
    dayTime = false;
    imgObject.src = "cloudy.jpg";
  }
  else {
    dayTime = true;
    imgObject.src = "sunny.jpg";
  }
}
```

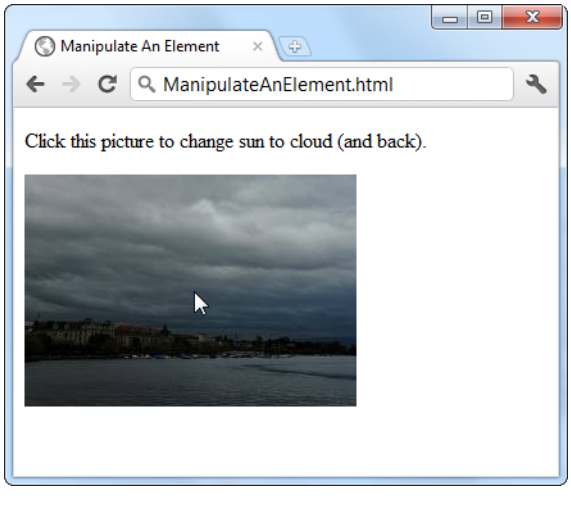


FIGURE B-3

Click this picture, and the page fires an event. That event triggers a function, and that function loads a new image.

Sometimes, an event passes valuable information to your event-handling function. To capture this information, you need to add a single parameter to your function. By convention, this parameter is usually named `event` or just `e`:

```
function swapImage(e) {
  ...
}
```

The properties of the event object depend on the event. For example, the onMouseMove event supplies the current mouse coordinates (which you'll use when creating the painting program on page 263).

There's one more fine point to note. When you use code to connect an event, you *must* put the entire event name in lowercase letters. This is different from when you wire up an event using an attribute in HTML. Unlike JavaScript, HTML doesn't care one whit about case.

NOTE This book refers to events using an easy-to-read convention called Pascal case, which uses uppercase letters to indicate each new word (for example, onLoad and onMouseOver). However, the code listings use all lowercase letters (for example, onload and onmouseover) because JavaScript requires it.

Inline Events

In order for the previous example to work, the swapImage() function must be defined somewhere else in your code. Sometimes, you may want to skip this step and define the function code in the same place where you attach the function to the event. This slick technique is called an *inline function*.

Here's an example that connects an inline function to the onClick event:

```
var imgObject = document.getElementById("dayImage");
imgObject.onclick = function() {
  // The code that went in the swapImage() function
  // now goes here.
  if (dayTime === true) {
    dayTime = false;
    imgObject.src = "cloudy.jpg";
  }
  else {
    dayTime = true;
    imgObject.src = "sunny.jpg";
  }
};
```

This shortcut approach to event handling is less common than using a separate, named function to handle the event. However, it's still a useful convenience, and the examples in this book use it occasionally.

NOTE Inline functions are sometimes useful when you're dealing with an *asynchronous task*—a task that the browser handles in the background. When an asynchronous task is finished, the browser fires an event to notify your code. Sometimes, the clearest way to deal with this situation is to put the code that handles the *completion* of a task right next to the code that triggered the *start* of the task. (Page 277 shows an example with a picture that's loaded asynchronously and then processed.)

Finally, there's one sort of inline function that's used in many of the examples in this book. It's the event handler for the window's onLoad event, which fires after the entire page is rendered, displayed, and ready to go. This makes it a logical point for your code to take over. If you try to run code before this point, you might run into trouble if an object hasn't been created yet for the element you want to use:

```
<script>
window.onload = function() {
  alert("The page has just finished loading.);
}
</script>
```

This approach frees you from worrying about the position of your script block. It lets you place the initialization code in the <head> section, where it belongs, with the rest of your JavaScript functions.

Index

Symbols

3-D transforms, 204
3D Walker website, 315
$_GET collection (PHP), 378–379
$_POST collection (PHP), 379
: (colon) and pseudo-classes, 443
{} (curly brackets), 464
== (equal to), 463
=== (exactly equal to), 463
@font-face, 206–208, 213–216
@font-face property, 438
> (greater than), 463
>= (greater than or equal to), 463
< (less than), 463
<= (less than or equal to), 463
&& (logical and), 463
|| (logical or), 463
@media block, 239–241
! (not), 463
!= (not equal to), 463
!== (not exactly equal to), 463
-prefix-free JavaScript tool, 185, 201
: (semicolon)
  JavaScript, 452
-webkit- vendor prefix, 183–184, 193–194, 197, 201

A

<abbr> element, 22
"About Me" page, retrofitting, 88–93
accessibility, 39
  best practices, 39
  canvas and, 274
  video captions, 169–176
<acronym> element, 22
<address> element, 24
<a> element, 24
Android
  animation performance, 303
  browser support for
    File API, 340
    forms, 119
    HTML5 audio formats, 153
    HTML5 video formats, 153
    IndexedDB, 353
    server-sent events, 387
    session history, 432
    <track> element, 174
    validation, 119
    web sockets, 394
    web workers, 425
  cache, 366
  calc() function, 226
  embedded font formats, 208
  H.264 Baseline Profile, 154
  linear-gradient() function, 193
  Miro Video Converter and, 156
  mobile-optimized video, 154
  simulators, 231
<applet> element, 22
applicationCache objects, 371–374
  swapCache() method, 374
  update() method, 374

ARIA (Accessible Rich Internet Applications), 82–83
<article> element, 21, 45–46, 76
  outlines and, 68–70
Arvidsson, Erik, 271
<aside> element, 21, 49–51, 56, 76
  <footer> element inside of, 63
  outlines and, 68–70
    solving problems with, 72–74
asynchronous tasks, 475
attribute values, 16
  quotation marks around, 17
Audacity, 156
audio
  Audacity, 156
  <audio>. *See* <audio> element
  automatic playback, 147
  browser support for, 150, 153–154
  codec, 151
  container format, 151
  controlling players with JavaScript, 160–169
  dynamically created or edited, 145
  editors, 156
  encoding media, 156
  fallbacks, 154–160
    Flash, 157–160
    supporting multiple formats, 155–156
  file formats
    MP3, 149–156
    Ogg Vorbis, 150, 153–154
    using multiple, 155–156, 163
    WAV, 147, 150, 153, 156
  Flash players, 157
  JavaScript and media players, 167–175
  licensed content, 145
  linking multiple media files together, 149
  looping playback, 147
  low-latency, high-performance, 145
  MIME type, 151
    why to use them, 152
  mobile browser support for, 153
  mute button, 147
  overview, 145–149
  paired with video, 147–149
  playback controls, 146
  preloading media files, 146–147
  recording, 145
  Web Audio API, 145
<audio> element, 21, 143
  adding sound effects, 161–164
  autoplay attribute, 147
  browser support for, 29
  controlling players with JavaScript, 160–169
    adding sound effects, 161–164
  controls attribute, 146
  empty syntax tags, 146
  example of, 145–146
  format fallbacks, 155–156
  hidden, 164
  loop attribute, 147
  mediagroup attribute, 149
  metadata, 146
  MIME type, 155
  multiple instances, 147, 163
  mute button, 147
  nested <source> elements, 155–156
  paired with <video> element, 147–149
  preload attribute, 146–147
  src attribute, 145–146
autocapitalize attribute, 122
autocomplete attribute, 122
autocorrect attribute, 122

B

background-color property, 438, 446
background-image property, 438
background-position property, 438
background-repeat property, 438
base-64 encoding, 269
<bdo> element, 21
<b> element, 23
Berners-Lee, Tim, 428
<big> element, 22
bitmap images, 258
Blackberry mobile-optimized video, 154
blank values and validation, 117
block elements, 51–52
Blu-ray players, 151, 153
<body> element, 11, 446
bold formatting, 23
bookmarklet, 66
border-color property, 438
border property, 438, 447

border-style property, 438
border-width property, 438
box shadows, 190–191
box-sizing, 226
browsers
blank values and validation, 117
browser-specific styles with CSS3, 183–184
clearing browser cache, 364
complex scripts and browser's automatic reconnection feature, 393
dealing with old, 27
differences in way rich HTML editing works in different, 138
extracting semantic data in, 92–93
feature detection, 32–34
finding requirements for HTML5, 27–29
GlobalStats, 30–31
Google Chrome. *See* Chrome
HTML5 and, 26–35
IE. *See* Internet Explorer
mobile. *See* Android; mobile browsers
Modernizr and, 32–34
Mozilla Firefox. *See* Firefox
old browsers and HTML5, 27
Opera. *See* Opera
placeholder text and, 110
plug-ins, 9
Safari. *See* Safari
single page for desktop browsers and mobile devices, 156
support for form validation, 119–122
Modernizr and, 120–121
polyfills and, 121–122
tricking into switching into XHTML mode, 21
turning into HTML editors, 136–140
used by audience, 30–31
using polyfills to fill HTML5 gaps, 35–36
web storage
communicating between different browser windows, 330–332
browser support for
Android
forms, 119
<audio> element, 29
canvas and, 271–274
File API, 340
foreign elements in Internet Explorer, 52
geolocation feature, 413
HTML5 audio formats, 150, 153
HTML5 video formats, 151, 153
IndexedDB, 353
new input types in forms, 124–125
offline applications, 364–366
semantic elements, 51–53
server-sent events, 387
session history, 432
<track> element, 174
validation, 119
video captions, 174–175
web sockets, 394
web storage, 325
web workers, 425

C

caching
bypassing cache when online, 370
clearing browser cache, 364
events, 372
practical techniques, 366–374
accessing uncached files, 366–368
adding fallbacks for cached files, 368–370
checking the connection, 370–371
updates with JavaScript, 371–374
query string and, 359
size, 366
tradition vs. offline applications, 357
triggering update for cached application, 363
with manifest files, 356–366
creating, 357–359
putting on web server, 359–362
updating, 362–364
using, 359
calc() function, 226
canvas, 245–274
accessibility, 274
animation, 300–307
basic, 301–302
for lazy people, 307
hit testing, 313–316

maze game, 307–316
multiple objects, 302–307
performance, 303
arc() method, 254–255
arcTo() method, 254–255
base-64 encoding, 269
beginPath() method, 251
bezierCurveTo() method, 254–255
bitmap images, 258
browser compatibility, 271–274
building a basic paint program, 263–271
drawing on the canvas, 266–268
preparing to draw, 264–266
saving the picture in canvas, 268–271
<canvas>. *See* <canvas> element
Canvas plug-in for Adobe Illustrator, 258
changeColor() function, 265
changeThickness() function, 266
chess simulator, 315
clearCanvas() function, 267
clearRect() method, 267, 298
closePath() medthod, 252–253, 255–256, 261
composite operations, 262–263
control points, 255–257
Core HTML5 Canvas, 315
createImageData() method, 276, 313
createLinearGradient() function, 285
createRadialGradient() function, 285
creating image object, 276
curved lines, 254–256
custom objects, 294–298
data URLs, 268–271
drawFrame() function, 302, 304, 305, 312
draw() function, 267
drawImage() function, 276–277
slicing, dicing, and resizing images, 278–279
drawing
for math-phobes, 258
graphs, 288–293
images, 276–277. *See* canvas, images
library, 258
text, 279–280
ExplorerCanvas library, 271–272
Fabric.js, 258
fillEllipse() method, 258
filling shapes with patterns, 283–284
fill() method, 252–253
fillRect() method, 253, 256–257
fillStyle property, 252–253, 283
getContext() method, 247
getElementById() method, 246
getElementById() method and, 246
getting started, 246–263
globalAlpha property, 261
gradients, 281, 284–288
addColorStop() method, 287
creating gradient object, 285
setting colors in, 287
graphs, 288–293
plotScore() function, 292
hit testing, 298–301, 313–316
HTML Canvas, 315
images
drawing, 276–277
slicing, dicing, and resizing, 278–279
squashed, 277
<img> element
creating image object, 276
handing data URL to, 269
onclick attribute, 265
iPaint, 271
isDrawing variable, 266–267
KineticJS, 258
lineTo() method, 249, 253, 267
making shapes interactive, 293–300
animation, 300–307
hit testing, 298–301
keeping track of what you've done, 294–298
matrix transform, 259
maze game, 307–316
animating the face, 311–313
drawMaze(), 309
hit testing, 313–316
processKey() function, 309–312
setting up, 309–310
Modernizr and, 273
moveTo() method, 249, 253
online examples, 315
onMouseDown event, 266, 267
onMouseMove event, 267

onMouseUp event, 264, 267
paint programs, 271
paths and shapes, 251–253
polyfilling
with ExplorerCanvas, 271–272
with FlashCanvas library, 272–273
processKey() function, 309–312
quadraticCurveTo() method, 254–255
requiring JavaScript, 245
restore() method, 260
rgba() function, 260
rgb() function, 250, 260
rotate transform, 259
save() method, 260
saving content in Firefox, 270
scale transform, 258
setTimeout() method, 302
shadowBlur property, 282
shadowColor property, 282
shadowOffsetX and shadowOffsetY properties, 282
shadows, 281–283
properties for creating, 282
Sketchpad, 271
Stack Overflow site, 258
startDrawing() function, 266
stopDrawing() function, 267
straight lines, 248–251
capping off ends, 250
lineTo() method, 249
moveTo() method, 249
setting color of, 250
stroke() method, 249
stroke() method, 251–263
strokeRect() method, 253
transforms, 256–260
documentation and examples, 260
translate transform, 258
transparency, 260–262
triangle-drawing code, 252
<video> element, 279
VML (Vector Markup Language), 271–272
Canvas Demos website, 315
<canvas> element, 21, 246, 248
fallback content, 273–274
squashed images, 277
Canvas-text JavaScript library, 272
capitalization, 16
Captionator.js, 175
** element,** 22
character encoding, 12–13
chess simulator, 315
Chrome
audio/video playback controls, 146
browser support for
File API, 340
HTML5 audio formats, 153
HTML5 video formats, 153
IndexedDB, 353
server-sent events, 387
session history, 432
<track> element, 174
validation, 119
web sockets, 394
web workers, 425
debugging JavaScript code, 462
extensions
h5o, 66
outline that lets visitors jump to the appropriate section in a page, 39, 65
Semantic Inspector, 93
HTML5 and, 26
Manual Geolocation plug-in, 404
transform property, 184
transforms, 201
vendor prefix, 183
WebVTT files and, 174
<cite> element, 24
CKEditor, 136
className property, 472
clear property, 438
client-side validation, 113
collapsible boxes, 60
colon (:) and pseudo-classes, 443
color
data type, 125, 129
gradients, 191–195, 250, 272
linear, 192–193
online gradient-generating tool, 195
radial, 193–194
repeating, 194–195
transitions, 198
property, 438
<command> element, 21, 135
composite operations, 262–263

conditional logic, 463–465
contenteditable attribute, 136–140
  using to edit element, 136–138
contextual selectors, 43, 442, 447–449
cookies, 319, 320, 325
coords object, 407–408
Core HTML5 Canvas, 315
createImageData() method, 276
Creating a Website: The Missing Manual, xiii, 196
CSS3
  background images, 188–190
  backwards compatibility, 179
  border-radius property, 179, 187
  box and text shadows, 190–191
  box-sizing, 226
  browser-specific styles, 183–184
  calc() function, 226
  creating fallbacks with Modernizr, 180–182
  current features, 178–184
  fonts. *See* web fonts
  gradients, 191–195
    linear, 192–193
    online gradient-generating tool, 195
    radial, 193–194
    repeating, 194–195
    transitions, 198
  matrix(n1, n2, n3, n4, n5, n6) function, 203
  modules, 177
  multicolumn text, 217–220
  responsive design. *See* responsive web design with CSS3
  rotate(angle) function, 203
  rounded corners, 187
  scaleX(x) function, 203
  scale(x, y) function, 203
  scaleY(y) function, 203
  shadows
    transitions, 198
  skewX(angle) function, 203
  skew(x-angle, y-angle) function, 203
  skewY(angle) function, 203
  specifications, 178
  transforms, 201–206
    3-D, 204
    shifting starting point, 204
    transform functions, 203
    transitions that use, 198, 204–206
  transitions, 195–206, 385
    basic, 196–198
    gradients, 198
    making more natural, 202
    shadows, 198
    that use transforms, 198, 204–206
    transparency, 198
  translateX(x) function, 203
  translate(x, y) function, 203
  translateY(y) function, 203
  transparency, 185–187
  vendor prefixes, 183–184
  when an em becomes a rem, 230
CSS3: The Missing Manual, xiii, 435
CSS (Cascading Style Sheets), 435–450
  <body> element, 446
  body rule, 446
  boxes, 440
  class attribute, 439
  class-specific rule, 446
  columns, 440
  comments, 439
   element, 440, 448
  elements
    naming, 439
  embedding style information directly into element, 435
  embedding style sheet in <style> element, 436
  example of style sheet magic, 436
  formatting right elements, 438–439
  header rule, 447
  ID selectors, 442–443
  linking separate style sheet file, 436
  media types, 234
  properties, 437, 437–438
    borders, 438
    colors, 438
    common, 438
    fonts, 438
    graphics, 438
    inherited, 440
    layout, 438
    overview of all, 438
    padding, 448
    size, 438
    spacing, 438
    text alignment, 438

selectors, 437, 446
attribute, 444
contextual, 441–442, 447, 448
creating, 439
ID, 442–443
multiple, 441
overlapping, 439
pseudo-class, 443–444
 element, 440, 448
styles, adding to page, 435–436
style sheets, 436–439
adding, 14–35
overview, 445–450
values, 437
inherited, 440
csszengarden.com, 436
curly brackets ({}), 464

D

databases. *See also* IndexedDB
difference between server-side and client-side, 341
primary keys, 346
<datalist> element, 21, 130–133
with <option> element, 131–133
data storage. *See* web storage
data URLs, 268–271
date data type, 125, 128–129
dates. *See* <time> element
datetime data type, 129
datetime-local data type, 129
debugging JavaScript code, 462
logical operators, 463
dedicated web servers, 395
designMode attribute, 136–140
editing pages, 138–140
<details> element, 21, 60
** element,** 37, 440, 448
in HTML, 40–43
in HTML5, 43
itemprop, itemscope, or itemtype attribute, 90–95
doctype, 11–12
standards mode, 12
document outlines. *See* outlines
drawing. *See* canvas
drawing library, 258

E

easier editing and maintenance, 38
echo server, 397–398
elements
adapted, 22–24
added, 21
removed, 22
standardized, 25–26
tweaked, 24–25
ellipsis in book examples, 41
email data type, 124–125
<embed> element, 21, 25
** element,** 23
em unit, 228
when an em becomes a rem, 230
enhanced search results, 94–98
EOT (Embedded Open Type), 207–208, 213
equal to (==), 463
events
caching, 372
exactly equal to (===), 463
ExplorerCanvas library, 271–272

F

Fabric.js, 258, 307
fallbacks
adding for cached files, 368–370
creating with Modernizr
CSS3, 180–182
JavaScript, 182
multiple background images, 190
feeds, 93
<fieldset> element, 107
<figcaption> element, 21, 48, 76
<figure> element, 21, 46–49, 76
File API, 332–340. *See also* FileReader object
browser support for, 340
getting hold of a file, 333
<input> element, 333
reading multiple files at once, 336–337
reading text file with, 333–336
reading image file wth drag-and-drop, 337–340
replacing standard upload control, 336

FileReader object, 334–340
abort() method, 340
drop() function, 339
ignoreDrag() function, 338
processFiles() function, 339
readAsArrayBuffer() method, 335
readAsBinaryString() method, 335
readAsDataUrl() method, 335
readAsDataURL() method, 337–340
readAsText() method, 335
Firefogg plug-in, 156
Firefox
audio/video playback controls, 146
browser support for
File API, 340
HTML5 audio formats, 153
HTML5 video formats, 153
IndexedDB, 353
server-sent events, 387
session history, 432
<track> element, 174
validation, 119
web sockets, 394
web workers, 425
calc() function, 226
debugging JavaScript code, 462
Firefogg plug-in, 156
HTML5 and, 26
metadata and <audio> element, 147
saving canvas content in, 270
transform property, 184
transforms, 201
vendor prefix, 183
Flash
fallbacks, 157–160
JavaScript media players, 168
Flowplayer Flash, 158
Flowplayer HTML5, 160
H.264 video file format, 151
players, 157
with HTML fallback, 160
FlashCanvas library, 272–273
Flash plug-ins, 9
float property, 438
Flowplayer Flash, 158
Flowplayer HTML5, 160
fluid design and media queries, 234–235
fluid images, 226–228
fluid layouts, 222–226. *See also* media queries
adapting with media queries, 231–244
max-width and min-width properties, 232
fluid typography, 228–230
<font> element, 22
font-family property, 438, 446
fonts
creating font collections, 216
embedded font formats, 208
licensing, 211
subscription sites, 216
web. *See* web fonts
font-size property, 438
Font Squirrel, 209–211
preparing fonts for web, 211–214
font-style property, 438
font-variant property, 438
font-weight property, 438
<footer> element, 21, 43–46, 50, 61–63, 76
animation, 62
close button, 61
fixed positioning, 61
in an <aside> element, 63
partially transparent background, 62
<form> element, 105–106, 114
forms, 103–140
autocapitalize attribute, 122
autocomplete attribute, 122
autocorrect attribute, 122
autofocus attribute, 111–112
bypassing form submission with JavaScript, 105
<command> element, 135
controls, 108
how browser draws, 109
<datalist> element, 130–133
with <option> element, 131–133
error-prevention and error-checking features, 124
<fieldset> element, 107
Google Instant feature, 105
HTML5Forms library, 122
<input> element, 107, 111, 115, 118
browser compatibility for new input types, 124–125
color data type, 129

date-related types, 128–129
email data type, 124–125
new data types, 123–130
number type, 126
range data type, 127–128
search data type, 126
tel data type, 126
url data type, 126
limit of, 109
<menu> element, 135
<meter> element, 133–135
mobile devices and, 124
multiple attribute, 122
overview, 104–105
placeholders, 109–111
browsers not supporting, 110
special characters and, 111
writing good, 111
placing controls outside of, 107
<progress> element, 133–135
revamping traditional forms, 105–112
spellcheck attribute, 122
starting in right place, 111–112
<textarea> element, 107, 111, 113, 117
validation, 112–119
browser support for, 119–122
client-side validation, 113
how HTML5 form validation works, 112–114
Modernizr, 120–121
polyfills and, 121–122
regular expression, 116–119
server-side validation, 113
styling hooks, 115–116
turning off, 114–115
XForms, 103
XMLHttpRequest object, 105, 133, 138, 140

frames feature, 22
Friedl, Jeffrey, 117
Fulton, Jeff, 315
Fulton, Steve, 315

G

Geary, David, 315
geolocation, 401, 402–413. *See also* Geolocation object
assessing accuracy of guess, 407
finding visitor coordinates, 405–406
how it works, 402–404
IP addresses, 402–403, 413
Manual Geolocation plug-in, 404
monitoring visitor's moves, 413
showing a map, 409–412
why you should use it, 404

Geolocation object
accuracy property, 407
browser compatibility and, 413
clearWatch() method (geolocation), 405, 413
coords object and, 405, 407–408, 412
enableHighAccuracy property, 409
errors, 406–408
getCurrentLocation() method, 405, 406, 408
getCurrentPosition() medthod, 405, 407, 409, 413
maximumAge property, 409
methods, 405
setting options, 408–409
timeout property, 409
watchPosition() method, 405, 413

GlobalStats, 30–31
global variables, 460
Goldwave, 156
Google
ignoring semantic data, 99
job search technology for veterans, 99
product searches, 99
Recipe View, 98–102
rich snippets, 94
Structured Data Testing Tool, 94–98

Google Analytics, 31, 368
Google Chrome. *See* Chrome
Google Fonts, 214–216
Google Instant feature, 105
Google Maps, 409–412
gradients, 191–195
linear, 192–193
online gradient-generating tool, 195
radial, 193–194
repeating, 194–195
transitions, 198

greater than (>), 463
greater than or equal to (>=), 463

H

h5o plug-in, 65
H.264 Baseline Profile, 154
H.264 video format, 149–160
converting to, 156
in Opera browser, 150
licensing, 154
HandBrake, 156
hardware acceleration, 303
hashbang URLs, 427–428
hCalendar microformat, 85
hCard microformat, 84
<head> element, 11
<header> element, 21, 43–46, 50, 76
adding headings you can't see, 55
getting IE to recognize, 52
multiple inclusions, 54
heading structure of a site, 56
height property, 438
highlighted text, 80–82
hit testing, 298–301, 313–316
<hr> element, 23
HTML
 element, 41–43
frames feature, 22
page structure the old way, 40–43
retrofitting traditional page, 39–50
HTML5
a living language, 6–7
availability, xiv–xv
back from the dead, 5
browser plug-ins, 9
browsers
finding requirements for, 27–29
used by audience, 30–31
Chrome and, 26
current, evolving draft of, 7
 element, 440
 element, 43
example, 15
Firefox and, 26
included features, 6
Internet Explorer and, 26
loosened rules, 16–17
number 5 in the name, 5
obsolete elements, 8
old browsers and, 27
Opera and, 26
Safari and, 26
smartphones and, 26
standard, 10
story of, 3–7
syntax, 16–21
tablet computers and, 26
three key principles of, 7–9
being practical, 9
"Don't break the Web", 7–8
"pave the cowpaths" approach, 8–9
using polyfills to fill HTML5 gaps, 35–36
viewing, xiii–xiv
vs. HTML, xv
writing, xiii
HTML5Forms library, 122
HTML Canvas, 315
HTML color code, 250, 446–447
HTML editors
browser differences, 138
turning browsers into, 136–140
<html> element, 11

I

<i> element, 23
<iframe> element, 22
image-based patterns, 250
images. *See also* <figure> element; <figcaption> element
background images with CSS3, 188–190
<img> element
handing data URL to, 269
onclick attribute, 265
reading image file wth drag-and-drop, 337–340
transparency, 185–187
transitions, 198
<img> element, 16
creating image object, 276
handing data URL to, 269
onclick attribute, 265
IndexedDB, 340–354
browser support for, 353
calling open() method, 344
creating and connecting to a database, 343–346
difference between server-side and client-side databases, 341
enhancing performance, 341

improving local storage, 341
key path, 345
LinkRecord object, 342–343, 347–348
making self-sufficient offline application, 341
naming database, 344
onError event handler, 344–345
onSuccess event handler, 344
onUpgradeNeeded event handler, 344–345
primary keys, 346
querying
all records in a table, 349–351
single record, 351–352
records
deleting, 352–353
querying all, 349–351
querying single, 351–352
similarities with local storage, 342
storing records in the database, 346–348
calling object store method, 347
creating transaction, 346
handling success and error events, 347
retrieving object store, 347
tables
creating, 345
querying all records in, 349–351
InfoWindow object, 412
inline functions, 475
innerHTML property, 472
<input> element, 21
File API, 333
reading text file with, 333–336
forms, 107, 111, 115, 118
browser compatibility for new input types, 124–125
color data type, 129
date-related types, 128–129
email data type, 124–125
new data types, 123–130
number type, 126
range data type, 127–128
search data type, 126
tel data type, 126
url data type, 126
multiple attribute, 122
reading multiple files at once, 336–337
"Insufficient data to generate the preview" error message, 96
interactivity elements, 21
Internet Explorer
alert( ) function, 453
audio/video playback controls, 146
browser support for
File API, 340
HTML5 audio formats, 153
HTML5 video formats, 153
IndexedDB, 353
server-sent events, 387
session history, 432
<track> element, 174
validation, 119
web sockets, 394
web workers, 425
calc() function, 226
debugging JavaScript code, 462
HTML5 and, 26
JavaScript patches that can bring IE up to speed, 110
radial-gradient() function, 184
storage events, 331
transform property, 184
transforms, 201
tricking into recognizing a foreign element, 52
vendor prefix, 183
workaround for <output> element, 79
iPad
cache, 366
Miro Video Converter and, 156
simulators, 231
IP addresses, 402–403, 413
iPaint, 271
iPhone
animation performance, 303
cache, 366
H.264 Baseline Profile, 154
Miro Video Converter and, 156
mobile-optimized video, 154
simulators, 231
italic formatting, 23
itemprop, itemscope, or itemtype attribute, 90–95
generating properly formatted microdata-enriched markup, 91
itemReviewed property, 96–97

J

JavaScript, 451–476. *See also* objects
abort() method, 340
adding, 14
appendChild() method, 473
arrays, 465–466
asynchronous tasks, 475
audio/video players, 160–169
adding sound effects, 161–164
creating custom video player, 164–167
beginPath() method, 251
bookmarklet, 66
buttons and, 108
bypassing form submission with, 105
calculations, 78–80
Canvas-text library, 272
Captionator.js, 175
checking whether browser is online, 370–371
clearWatch() method, 405, 413
client-side code, 106
closing </script> tag, 14
collapsible boxes, 60
conditional logic, 463–465
connecting to event dynamically, 473–475
contenteditable, 137
datalist and, 133
dealing with old browsers, 27
debugging code, 462
defining data structure in, 342
drawing. *See* canvas
dynamically connecting to events, 470–471
embedding script in markup, 452–453
events. *See* JavaScript events
extending history object, 425–432
Fabric.js, 258, 307
fallbacks, 180–182
multiple background images, 190
transparency, 186
focus() method, 111
freeze-up problem, 414
functions. *See* JavaScript functions
geolocation, 405–413
getAttribute() method, 473
getCurrentLocation() method, 405–408
getCurrentPosition() method, 405, 407, 409, 413, 470
getElementById() method, 471
hiding element that wraps footer, 61
how web pages use, 452–459
HTML5 features that require, 375
inline events, 475–476
insertChild() method, 473
interacting with offline applications, 371–374
interacting with pages, 470–476
jQuery UI, 201
JS API group, 28
KineticJS, 258, 298, 307
language="JavaScript" attribute, 14
logical operators, 463
loops, 465
manipulating elements, 471–473
media players, 167–169
controlling with JavaScript, 160–169
creating custom, 164–167
Flash fallbacks, 168
Microdata Tool, 92
Modernizr tool. *See* Modernizr
MooTools, 201
moving code to script file, 455–457
null values, 460
Paper.js, 307
patches that can bring IE up to speed, 110
peeking at the DOM, 11
polyfill that adds autofocus support, 112
postMessage() method, 418, 421–423
web workers and, 418
-prefix-free, 185, 201
processFiles() function, 339
properties and HTML5 specification, 117
pushState() method, 426, 429–430
readAsArrayBuffer() method, 335
readAsBinaryString() method, 335
readAsDataUrl() method, 335
readAsText() method, 335
reading text files, 333–336
regular expressions, 116
removeChild() method, 473
replaceState() method, 431
responding to events, 457–459

retrieving data from server, 470
RGraph, 293
<script> block, 452
</script> tag and, 456
semicolon (:), 452
sending data to the server, 470
setAttribute() method, 473
setCustomValidity() method, 117–119
terminate() method, 421
transitions, 198–220
adding, 385
tricking IE into recognizing a foreign element, 52
updating pages, 470
variables, 459–460
data types, 461
global, 460
local, 460
naming, 457, 459
null values, 460
operations, 461–463
scope, 460
VideoJS player, 167–169
watchPosition() method, 405, 413
web storage. *See* web storage
XMLHttpRequest objects. *See* XMLHttpRequest objects
XML parser, 335
ZingChart, 293
JavaScript events, 455
common object events, 458–459
connecting dynamically to, 473–475
event handlers, 457, 473, 476
event property approach, 473
responding to, 457–459
server-sent, 387
JavaScript functions, 454–455
adding code to, 454
adding events, 455
advantages of using, 455
alert(), 452, 454, 467
arguments, 467
boing(), 163
calling, 454–455
doSearch(), 416–432
drop(), 339
findPrimes(), 416, 423
goToNewSlide(), 384, 430
goToNextSlide(), 432
ignoreDrag(), 338
importScripts(), 420
inline, 475, 475–476
linear-gradient(), 193
LinkRecord(), 347
naming, 457
Number(), 325, 328–329
object-definition, 468–469
processFiles(), 334, 336, 339
receiving and returning data, 466–467
rgb(), 446–447
canvas, 250, 260
rgba(), 260
setInterval(), 386
animation, 301
web workers and, 414, 424–425
setTimeout(), 386
animation, 301
web workers and, 414, 424–425
showFileInput(), 336
showMessage(), 454–457
parameters, 466–467
swapImage(), 473, 475
that receive and return data, 466–467
triggering, 454
using, 454–455
JavaScript & jQuery: The Missing Manual, xiii, 451
Java web socket server, 399
job search technology for veterans, 99
jPlayer, 167
jQuery UI, 201
JSON (JavaScript Object Notation), 329–330

K

Kaazing web socket server, 399
<keygen> element, 21
KineticJS, 258, 298, 307

L

LAME MP3 encoder, 156
lang attribute, 13
LatLng object, 412
left property, 438
less than (<), 463
less than or equal to (<=), 463

letter-spacing property, 438
licensed content, 145
line caps, 250
line-height property, 438
<link> element, 436
LinkRecord object, 342–343, 347–348
local variables, 460
logical and (&&), 463
logical operators, 463
logical or (||), 463
loops, 465

M

<main> element, 63–65, 76
manifest files, 356–366
  adding fallbacks for cached files, 368–370
  browser support for offline applications, 364–366
  caching and query string, 359
  clearing browser cache, 364
  creating, 357–359
  putting on web server, 359–362
  triggering update for cached application, 363
  updating, 362–364
  using, 359
Manual Geolocation plug-in, 404
margin-bottom property, 438
margin-left property, 438
margin property, 438
margin-right property, 438
margin-top property, 438
<mark> element, 21, 80–82
mark of the Web comment, 14
Mastering Regular Expressions, 117
max-device-width media feature, 233, 235, 238, 242–244
max-width media feature, 233–242
McFarland, David Sawyer, xiii, 435, 451
media. *See* audio; video
media queries, 221
  adapting layout with, 231–244
  anatomy of, 233–234
  building mobile-friendly layout, 236–239
  creating simple, 234–235
  for video, 244
  max-device-width media feature, 233, 235, 238, 242–244
  max-width media feature, 233–242
  @media block, 239–241
  more advanced query conditions, 239–241
  most useful features for building, 233–234
  orientation media feature, 234–235, 238, 242–244
  recognizing specific mobile devices, 242–244
  replacing entire style sheet, 241–242
<menu> element, 21, 135
metadata
  <audio> element, 146
  enhanced search results, 94–98
  how search engines use, 93–102
    enhanced search results, 94–98
    rich snippets, 94
  plug-ins that can spot different types of, 92
<meta> element, 231, 236
<meter> element, 21, 133–135
microdata, 75, 85–88
  formats, 98
  generating properly formatted microdata-enriched markup, 91
  namespaces, 86
  nested structure, 90
  Recipe data format, 98–102
  vs. micro formats, 87
  when to define new section, 91
Microdata Tool, 92
microformats, 84–85
  Recipe data format, 98–102
  vs. microdata, 87
MIME type, why to use them, 152
Miro Video Converter, 156
mobile browsers. *See also* Android
  audio and video support, 153
  H.264 Baseline Profile, 154
  mobile-optimized video, 154
  single page for desktop browsers and mobile devices, 156
mobile devices. *See also* Android; iPad; iPhone; mobile browsers
  common device widths, 243
  forms and, 124
  recognizing specific, 242–244

Modernizr, 32–34, 51, 53
canvas and, 273
creating fallbacks with
CSS3, 180–182
JavaScript, 182
form validation and, 120–121
full list of features, 32
hiding and replacing sections, 240
month data type, 125, 129
MooTools, 201
MP3 file format, 149–156
LAME MP3 encoder, 156
multicolumn text, 217–220
multiple attribute, 122

N

natural language, 13
navigator object, 405
<nav> section, 21, 39, 50, 56–59, 76
multiple, 58
outlines and, 68–70
within <aside> or <header>, 59
nested structure and microdata, 90
.NET web socket server, 399
<nobr> element, 26
node.JS web socket server, 399
not (!), 463
not equal to (!=), 463
not exactly equal to (!==), 463
null values, 460
number data type, 125, 126
numeric variables, 461–476

O

<object> element, 22
objects, 468–469. *See also* FileReader object; Geolocation object; XMLHttpRequest objects
applicationCache, 371–374
swapCache() method, 374
update() method, 374
common events, 458–459
common properties, 472
coords, 407–408
custom objects in canvas, 294–298
defining, 468
InfoWindow, 412
JSON (JavaScript Object Notation), 329–330
LatLng, 412
LinkRecord, 342–343, 347–348
literals, 469–470
naming, 457
navigator, 405
storing, 329–330
WebSocket
creating, 395–397
detecting failed connection, 397
socket-connection code, 397
offline application feature, 355–374
browser support for, 364–366
browser support for offline applications, 364–366
bypassing cache when online, 370
caching and query string, 359
caching events, 372
manifest files, 356–366
clearing browser cache, 364
creating, 357–359
putting on web server, 359–362
updating, 362–364
using, 359
practical caching techniques, 366–374
accessing uncached files, 366–368
adding fallbacks for cached files, 368–370
checking the connection, 370–371
updates with JavaScript, 371–374
triggering update for cached application, 363
troubleshooting, 362
when to work offline, 356
Ogg Theora video file format, 151, 153
converting to, 156
Ogg Vorbis audio file format, 150, 153–154
** element,** 25
onBlur event, 458
onCached event, 372
onChange event, 459
onChecking event, 372
onClick event, 458
onDownloading event, 372
onError event, 372, 459
onFocus event, 458
onKeyDown event, 458
onKeyUp event, 458
Online HTML outliner, 65

onLoad event, 459
onMouseDown event, 266–267
onMouseMove event, 267
onMouseOut event, 458
onMouseOver event, 457–458, 475
onMouseUp event, 264, 267
onNoUpdate event, 372
onObsolete event, 372
onPopState event, 429, 430, 431
onProgress event, 372
onSelect event, 459
onUnload event, 459
onUpdateReady event, 372
Opera
browser support for
File API, 340
HTML5 audio formats, 153
HTML5 video formats, 153
IndexedDB, 353
server-sent events, 387
session history, 432
<track> element, 174
validation, 119
web sockets, 394
web workers, 425
debugging JavaScript code, 462
extension for outlines, 66
H.264 video, 150
HTML5 and, 26
transform property, 184
transforms, 201
vendor prefix, 183
operations, 461–463
optional word break, 25
<option> element with <datalist> element, 131–140
orientation media feature, 234–235, 238, 242–244
OTF (OpenType PostScript), 207–208, 210
outlines, 65–74
basic, 66–68
sectioning elements, 68–70
solving an outline problem, 70–74
viewing, 65–66
<output> element, 21, 78–80
Internet Explorer workaround, 79

P

padding-bottom property, 438
padding-left property, 438
padding properties, 448
padding property, 438
padding-right property, 438
padding-top property, 438
page redirect, 432
page structure, 37–74, 75
elements, 76
nested structure and microdata, 90
the old way, 40–43
page structuring elements, 21
Paper.js, 307
parentElement property, 472
Person data format, 98
PHP scripts
$_GET collection, 378–379
$_POST collection, 379
asking the web server a question, 377–382
calling the web server with, 380–382
close() method of the EventSource object, 391
complex scripts and browser's automatic reconnection feature, 393
creating, 378–379
dot operator (.), 390
final argument of the open() method, 380
flush() function, 390
onMessage event, 390
polling with server-side events, 392
sending messages with a server script, 388–390
time() function, 389
PHP web socket server, 399
polling with server-side events, 386, 391–393
polyfills, 35–36
canvas
with ExplorerCanvas library, 271–272
with FlashCanvas library, 272–273
form validation and, 121–122
position property, 438
primary keys, 346

prime numbers that fit in certain range, 414–432
<progress> element, 21, 133–135
playback progress bar, 166
pseudo-classes
for transitions, 195, 196, 198
validation styling hooks, 115
pull-quotes, 49–50, 56, 70, 74
Python web socket server, 399

Q

query string, 379–380
quotation marks around attribute values, 17

R

range data type, 125, 127–128
Rating data format, 98
RDFa (Resource Description Framework), 83
Recipe data format, 99–102
Recipe View (Google), 98–102
recording audio or video, 145
regular expression and form validation, 116–140
rems, 230
responsive web design with CSS3, 221–244. *See also* media queries
basics, 222–231
box-sizing, 226
calc() function, 226
fluid images, 226–228
fluid layouts, 222–226
fluid typography, 228–230
making layouts work on smartphones, 230–231
max-width and min-width properties, 232
when an em becomes a rem, 230
retrofitting an "About Me" page, 88–93
reviewBody property, 96–97
Review data format, 98
rgba() function and transparency, 186
rgb() function, 250, 446–447
RGraph, 293
rich snippets, 94
right property, 438
rounded corners, 187
<rp> element, 21
RSS feeds, 93
<rt> element, 21
<ruby> element, 21
Ruby web socket server, 399
running applications
offline. *See* offline application feature

S

Safari
browser support for
File API, 340
HTML5 audio formats, 153
HTML5 video formats, 153
IndexedDB, 353
server-sent events, 387
session history, 432
<track> element, 174
validation, 119
web sockets, 394
web workers, 425
debugging JavaScript code, 462
HTML5 and, 26
linear-gradient() function, 193
transform property, 184
transforms, 201
vendor prefix, 183
screen reader, 39
<script> block, 452
</script> tag and, 456
script files, moving code to, 455–476
search data type, 124, 126
search engine optimization (SEO), 39, 93
hashbang URLs and, 428
search engines
enhanced search results, 94–98
Recipe View (Google), 98–102
smarter search filtering, 98
Structured Data Testing Tool, 94–95
using metadata, 93–102
video subtitles and, 173
<section> element, 21, 59–60, 76
outlines and, 68–70
sectioning elements, 68–70
complex pages and, 71
sections, hiding and placing, 240
<select> element, 122

selectors, contextual, 43, 442, 447–449
<s> element, 23
semantic data. *See also* metadata; microdata; microformats
  extracting in browser, 92–93
  Google ignoring, 99
  hidden, 99
  structuring pages with. *See* page structure
  text-level information, 75–102
semantic elements, 38–39
  browser compatibility for, 51–53
  designing a site with, 53–65
  how they were chosen, 50
  styling, 51
Semantic Inspector, 93
semicolon (:) in JavaScript, 452
server-side programming. *See also* PHP scripts; web servers; web sockets; XMLHttpRequest objects
  checking for server-side event support, 391
  complex scripts and browser's automatic reconnection feature, 393
  final argument of the open() method, 380
  frameworks, 55
  processing messages in web page, 390–391
  sending messages with a server script, 388–390
  server-sent events, 375, 386–393
    browser support for, 387
    message format, 387
  server-side events
    polling with, 391–393
  server-side includes
    session history and, 431
  validation, 113
  XMLHttpRequest objects, 105, 133, 138, 140, 430
    web workers and, 424
session history, 401, 425–432
  back() method, 426
  browser compatibility and, 432
  browser support for, 432
  forward() method, 426
  go() method, 426
  goToNewSlide() function, 430
  hashbang URLs, 427–428
  onPopState event, 429, 430, 431
  pushState() method, 426, 429–430
  replaceState() method, 431
  server-side includes and, 431
  templates and, 431
  three basic methods, 425
  URL problem, 426
sidebars, 56–59. *See also* <aside> element
  complex, 58
  shaping into three sections, 57
Silverlight, 151
simulators, 231
Sketchpad, 271
<small> element, 22
smartphones
  HTML5 and, 26
  making layouts work on, 230–231
<source> element, 21
  media attribute, 244
** element,** 41, 440, 448
  itemprop, itemscope, or itemtype attribute, 90–95
special characters, 111
spellcheck attribute, 122
Stack Overflow site, 258
standards that boost semantics, 82–88
  ARIA (Accessible Rich Internet Applications), 82–83
  microdata, 85–88
    namespaces, 86
    vs. microformats, 87
  microformats, 84–85
  RDFa (Resource Description Framework), 83
streaming video, 145
<strike> element, 22
** element,** 23
Structured Data Testing Tool, 94–98
  HTML tab, 95
  URL tab, 95
<style> element, 436
style property, 472
<summary> element, 21, 60
SVG (Scalable Vector Graphics), 207–208, 213

T

tablet computers and HTML5, 26
tagName property, 472
tel data type, 124, 126
templates, 55
  session history and, 431
text
  drawing in canvas, 279–280
  highlighted, 80–82
  multicolumn, 217–220
  optional word break, 25
text-align property, 438
<textarea> element, 107, 111, 113, 117
text-decoration property, 438
text elements, 21
text-indent property, 438
text-level information, 75–102
text shadows, 190–191
Thing data format, 98
Third Edition, xiii
time-consuming tasks, 414–417
time data type, 125, 129
<time> element, 21, 38, 77–78
TinyMCE, 136
<track> element, 172–173
  browser support for, 174
transforms, 201–206
  3-D, 204
  shifting starting point, 204
  transform functions, 203
  transitions, 198, 204–206
transitions
  CSS3, 195–206, 385
    basic transition, 196–198
  gradients, 198
  JavaScript, 385
  making more natural, 202
  pseudo-classes for, 195, 196, 198
  shadows, 198
  transforms, 198, 204–206
  transparency, 198
  triggering with JavaScript, 198–220
transparency, 185–187, 260–262
  fallbacks, 186
  rgba() function, 186
  transitions, 198
troubleshooting
  debugging JavaScript code, 462
  "Insufficient data to generate the preview" error message, 96
  offline application feature, 362
<tt> element, 22
TTF (TrueType), 207–208, 210

U

url data type, 124, 126
URLs
  changing to anything you like. *See* session history
  creating extra pages to satisfy, 431
  data URLs, 268–271
  hashbang, 427–428
UTF-8 encoding, 13

V

validation, 17–19
  blank values and, 117
  browser support for, 119
  forms, 112–119
    browser support for, 119–122
    client-side validation, 113
    how HTML5 form validation works, 112–114
    Modernizr, 120–121
    polyfills and, 121–122
    regular expression, 116–119
    server-side validation, 113
    styling hooks, 115–116
    turning off, 114–115
  JavaScript setCustomValidity() method, 117–119
  potential problems that a validator can catch, 17
  validator provided by the W3C standards organization, 17–19
  XHTML5 validator, 20
vendor prefixes, 183–184
  -prefix-free, 185
  -webkit-, 183–184, 193–194, 197, 201
veterans, job search technology for, 99
video. *See also* <video> element
  accessibility. *See* video, subtitles
  adaptive streaming, 145
  automatic playback, 147
  browser support for, 153–154
    captions, 174–175
    HTML5 formats, 151, 153
  captions. *See* video, subtitles
  codec, 149, 151
  container format, 151

drawing video frame, 279
encoding media, 156
fallbacks, 154–160
  Flash, 157–160
  Flash player with HTML fallback, 160
  format, 154
  supporting multiple formats, 155–156
  technology, 154
file formats
  encoding media, 156
  H.264, 149–160
  Ogg Theora, 151, 153
  using multiple, 155–156
  WebM, 151, 153–156, 158, 160
Firefogg plug-in, 156
Flash players, 157
  with HTML fallback, 160
Flowplayer Flash, 158
Flowplayer HTML5, 160
frame, 148
H.264 Baseline Profile, 154
H.264 licensing, 154
HandBrake, 156
JavaScript
  Captionator.js, 175
  controlling players with, 160–169
  creating custom player with, 164–167
  media players, 167–169
jPlayer media player, 167
licensed content, 145
linking multiple media files together, 149
link to download file and open it in an external program, 158
looping playback, 147
media queries for, 244
MIME type, 151
  why to use them, 152
Miro Video Converter, 156
mobile browser support for, 153
mobile-optimized, 154
overview, 145–149
paired with audio, 147–149
playback controls, 146
  H.264 video in Opera, 150
playback progress bar, 166
preloading, 146–147
recording, 145
single page for desktop browsers and mobile devices, 156
subtitles, 169–175
  adding, 172–173
  search engines and, 173
  timed text tracks, 170
  versus captions, 173
  WebVTT file, 170–176
VideoJS player, 167–169
YouTube, 157
Zencoder, 156
<video> element, 21, 38
  adding subtitles with <track> element, 172–173
  autoplay attribute, 147, 148
  controlling players with JavaScript, 160–169
  controls attribute, 146, 148
  drawing video frame, 279
  empty syntax tags, 146
  format fallbacks, 155–156
  height attribute, 148
  inserting Flowplayer Flash, 158
  loop attribute, 147, 148
  mediagroup attribute, 149
  metadata, 146
  MIME type, 155
  muted attribute, 148
  nested <source> elements, 155–156
  paired with <audio> element, 147–149
  poster attribute, 148–149
  preload attribute, 146, 148
  <progress> element
    playback progress bar, 166
  src attribute, 146, 148
  width attribute, 148
VideoJS, 167–169
Vimeo, 151
VML (Vector Markup Language), 271–272
void element, omitting final slash, 16

W

W3C validator, 18–19
WAI-ARIA, 83
WAI (Web Accessibility Initiative) website, 39

watchPosition() method (geolocation), 405, 413
watermark, 109
WAV audio file format, 147, 150, 153, 156
<wbr> element, 21, 25
Web Audio API, 145
web fonts, 206–220
converting desktop fonts to, 209
creating font collections, 216
embedded font formats, 208
finding, 208–209
@font-face, 206–208, 213–216
Font Squirrel, 209–211
preparing fonts for web, 211–214
formats, 207–208
free, 209–211
Google Fonts, 214–216
licensing, 211
preparing fonts for web, 211–214
some problems with, 209
subscription sites, 216
using on computer, 211
web forms, 21
WebM video file format, 151, 153–156, 158, 160
converting to, 156
web pages
heading structure, 56
JavaScript and, 452–459
retrofitting an "About Me" page, 88–93
sections, hiding and placing, 240
turning into websites, 55
web servers, 375–400. *See also* server-side programming; web sockets
asking a question, 377–382
calling the web server, 380–382
creating PHP script, 378–379
dedicated, 395
history of, 376
polling, 386
sending messages to, 376–386
time, 388
XMLHttpRequest objects, 105, 133, 138, 140, 430
web workers and, 424
WebSocket objects
creating, 395–397, 397
detecting failed connection, 397
socket-connection code, 397
web sockets, 375, 393
browser support for, 394
checking support for, 394
echo server, 397–398
examples on the web, 397–400
simple client, 395–397
web socket server, 394–395
web socket servers, 399
Web SQL Database, 353
web storage, 319–354. *See also* File API; IndexedDB
application preferences, 325
application state, 319, 325
basics, 320–325
browser support for, 325
communicating between different browser windows, 330–332
cookies, 319–320, 325
finding all stored items, 326–327
JSON (JavaScript Object Notation), 329–330
local storage, 319–321
improving, 341
storing data, 321–323
session storage, 320–321
storing data, 321–323
storage events, 331
storage space limits, 321
storing last position in a game, 323–325
storing numbers and dates, 327–328
storing objects, 329–330
user preferences, 319
without web server, 323
WebVTT file, 171–176
Chrome and, 174
web workers, 401
accessing code in another JavaScript file, 420
browser support for, 425
cancelling background task, 421
communicating with web pages, 418
creating multiple, 424
doing periodic tasks with, 424
downloading data with, 424
error handling, 421

nested, 424
passing complex messages, 421–424
postMessage() method, 418, 421–423
prime numbers that fit in certain range, 414–425
reusing for multiple jobs, 424
running offline, 420
safety measures, 415
sending progress percentage to page, 422
setInterval() function and, 414, 424–425
setTimeout() function and, 414, 424–425
time-consuming tasks and, 414–417
Worker object, 417–420
XMLHttpRequest object, 424
week data type, 125, 129
WHATWG (Web Hypertext Application Technology Working Group), 5–6
white-space property, 438
width property, 438
Wikipedia knowledge map website, 315
WOFF (Web Open Font Format), 208
word-spacing property, 438
Worker object, 417–420

X

XForms, 103
XHTML
enforcing syntax rules, 20
return of, 19–21
tricking browser into switching into XHTML mode, 21
version 1.0, 4
version 2, 4–5
XHTML5 validator, 20
XMLHttpRequest objects, 105, 133, 138, 140, 377, 430
asking the web server a question, 377–382
calling the web server, 380–382
getting new content, 382–386
goToNewSlide() function, 384
onReadyStateChange event, 380–381
query string, 379
sending messages to web server, 376
send() method, 381, 396
web workers and, 424

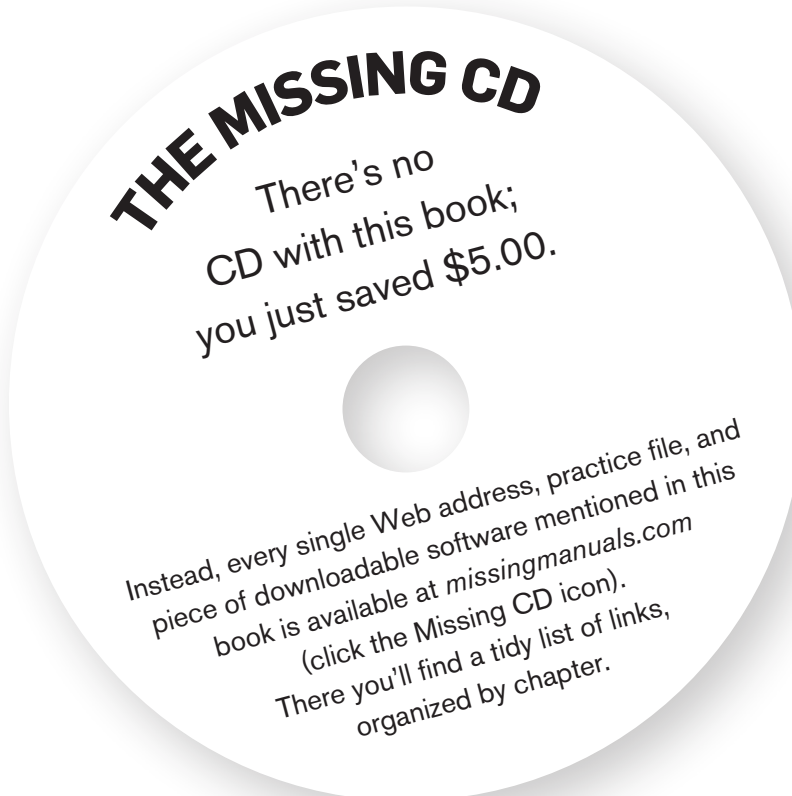
Y

Your Brain: The Missing Manual, 290
YouTube
H.264 video file format, 151
re-encoding videos, 157
screen reader video on, 39
trial HTML5 player, 144

Z

Zencoder, 156
ZingChart, 293

HTML5



Don't miss a thing!
Sign up for the free Missing Manual email announcement list at missingmanuals.com. We'll let you know when we release new titles, make free sample chapters available, and update the features and articles on the Missing Manual website.