Add Interactivity to Your Site

mini

the missing manual®

O'REILLY®

Matthew MacDonald

Add Interactivity to Your Site: The Mini Missing Manual
by Matthew MacDonald

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

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

January 2010: First Edition.

ISBN: 9781449382513

Table of Contents

Introduction v

Add Interactivity To Your Site 1

Understanding JavaScript 1

Server-Side and Client-Side Programming 3

Scripting Languages 5

JavaScript 101 5

The <script> Element 5

Variables 12

Functions 18

External Script Files 24

Dynamic XHTML 26

XHTML Objects 27

Events 36

Image Rollovers 40

Collapsible Text 42

An Interactive Form 46

Scripts on the Web 50

Finding a Cool Script 53

Colophon 59

Introduction

The days of Web pages that have nothing more than text and a picture or two are long gone. To be competitive in today's Internet landscape, you need pages that pop. This Mini Missing Manual shows you how to add interactive elements to your pages: menus that expand when a visitor clicks on them, images that change with a mouse roll, and pages that respond to information people type into forms.

The key to pages like these is JavaScript, a simple programming language that's surprisingly powerful. This book gives you enough background on JavaScript so you can find great (and free) JavaScript code online, understand it well enough to make basic changes, and then to paste it into your pages to get the results you want.

> ***Tip:*** To learn more about building and improving your site, see *Creating a Web Site: The Missing Manual*.

Add Interactivity To Your Site

JavaScript is a simplified programming language designed to beef up your website pages with interactive features. It gives you just enough programming muscle to add some fancy effects, but not enough to cause serious damage if your code goes wonky. It's perfect for creating pop-up windows, marquee-style scrolling text, and buttons that light up when visitors mouse over them.

The goal of this book isn't to teach you all the details of JavaScript—instead, it's to give you enough background so you can find great JavaScript code online, understand it well enough to make basic changes to the code, and then to paste it into your pages to get the results you want. Since the Web has dozens of JavaScript sites offering thousands of ready-made scripts for free, these basic skills can come in very handy.

Understanding JavaScript

The JavaScript language has a long history—it first hit the scene with the Netscape Navigator 2 browser in 1995, and Internet Explorer jumped on the bandwagon by adding JavaScript compatibility to IE version 3. Today, all modern browsers support JavaScript, and it's become wildly popular as a result. However, some justifiably paranoid web visitors turn off JavaScript compatibility

because malicious developers have, on occasion, used JavaScript-fueled agents to attack computers with pop-up ads and other types of browser annoyances. That means that the best rule of thumb with JavaScript is to use it to improve your pages, but make sure your page still works (even if it doesn't look quite as nice) for people who disable it.

> ***Note:*** JavaScript is thoroughly different from the programming language called Java (although the code sometimes looks similar, because they share some syntax rules). Sun Microsystems developed Java, and it's a full-fledged language, every bit as powerful—and complicated—as C++, C#, and Visual Basic.

So what can JavaScript do?

- It can dynamically insert new content into a web page or modify an existing web page element. For example, you can display a personalized message to your visitors ("Hello, Joe!") or make titles grow and shrink perpetually (an example of which is shown in "Using XHTML objects in a script" below).
- JavaScript can gather information about the current date, your visitor's browser, or the characters your visitors type into a form you include on a web page. You can display any of this information on a page or use it to make decisions about what your site does next. For example, you could stop visitors from going any further in your site until they type in an email address.
- JavaScript can react to events that take place in a browser. For example, you can write code that runs when a page finishes loading or when a visitor clicks a picture.

It's just as important to understand what JavaScript *can't* do. JavaScript code is *sandboxed*, which means a browser locks your page into a carefully controlled place in memory (known as a sandbox) so it can't access anything on your visitor's computer. This design, which is necessary to ensure good security, effectively prevents JavaScript from performing any potentially risky tasks, like sending

orders to a printer, creating or editing files on a hard drive, running other programs, reformatting a drive, and so on. In fact, about the only thing JavaScript *can* do is modify the appearance of your web page.

Server-Side and Client-Side Programming

To understand how JavaScript fits into the web universe, it's important to understand the two types of programming on the Web.

When you use a search engine like Google or go to an e-commerce site like Amazon, you're actually connecting to a high-powered piece of software, known as a *server-side application*, which runs on a web server. When you visit one of these sites, you send information to this server-side program, information like the keywords you want to search for or the book you want to buy. The program, in turn, consults a massive database and spits out some XHTML (the code that creates web pages), which then generates the page you see in your browser.

Server-side applications rule the web world, because there's virtually nothing they can't do. However, they're insanely difficult to program. Not only do developers need to worry about getting the program to generate XHTML for a browser, they also need to make sure the program can run all kinds of complex routines and consult giant databases—and they need to do it all in a way that performs just as well when millions of people are clamoring for attention as it does when there's only one person visiting a site. This is hard work, and it's best handled by the poor souls we call programmers.

Client-side applications, on the other hand, use a completely different model. They embed small, lightweight programs inside an ordinary XHTML page. When a browser downloads the page, the browser itself runs the program (assuming security settings or compatibility issues haven't disabled it). Client-side programs are much less powerful than those on the server side—they can't

reliably access the huge databases stored on web servers, for example, and for security reasons they can't directly change most things on your home computer. However, they're much simpler to create. If you've ever played a game of Java checkers in your browser (see Figure 1-1), you've used a client-side program.

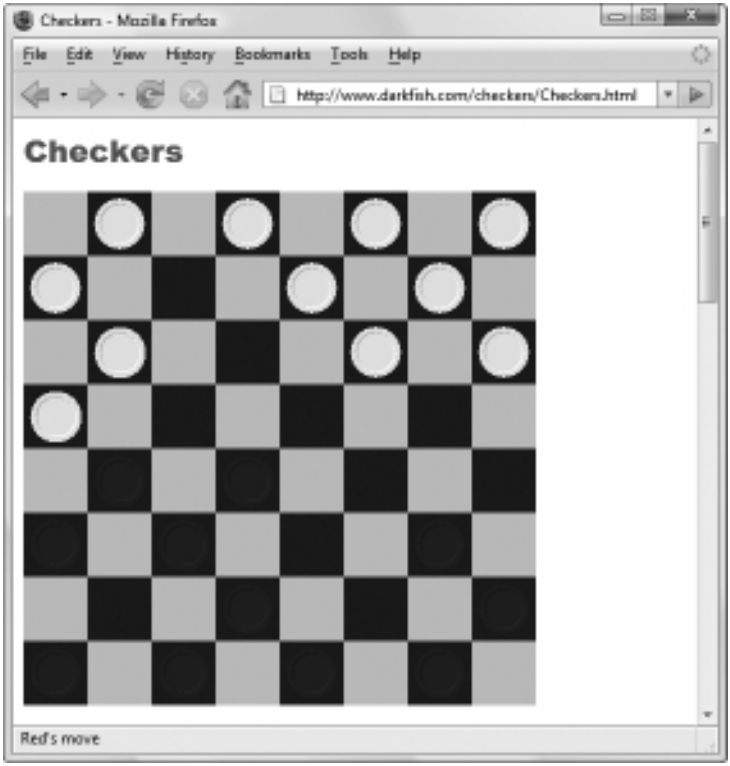

Figure 1-1: *This game of Java checkers looks like an ordinary web page, but something very different is taking place behind the scenes. In fact, the checkerboard is actually a complete Java program embedded inside the web page.*

Scripting Languages

Even client-side programs can be a challenge for non-technogeeks. For example, to build the checkers game shown in Figure 1-1, you need to know your way around the highly sophisticated Java programming language. However, a whole class of client-side programs exist that aren't nearly as ambitious. They're called *scripts*, and their purpose in life is to give a browser a list of instructions, like "scroll that heading from left to right" or "pop up an ad for fudge-flavored toothpicks in a new window."

Scripts are written in a simplified *scripting language*, and even if you don't know all the ins and outs of a language, you can often copy and paste a cool script from a free website to get instant gratification. Two examples of scripting languages are JavaScript and VBScript (the latter a scripting language that uses syntax resembling Visual Basic).

This chapter focuses exclusively on scripts you create with JavaScript, the only scripting language that's reliably supported on most browsers.

JavaScript 101

Now that you've learned a bit about JavaScript and why it exists, it's time to dive in and start creating your first real script.

The <script> Element

Every script starts with a <script> block you slot somewhere into an XHTML document. Really, you have only two options:

- **The <body> section.** Put scripts that you want your browser to run right away in the <body> section of your XHTML. The browser launches your script as soon as it reaches the <script> element. If you put your script at the beginning of the <body> section, your browser fires up the script before it displays the page.

Tip: Usually, JavaScript fans put their scripts at the *end* of the <body> section. That way, you avoid errors that might occur if you use a script that relies on another part of the page, and the browser hasn't read that other section yet. Because browsers read an entire page quickly, these scripts execute almost immediately.

- **The <head> section.** If you place an ordinary script in the <head> section of your XHTML document, it runs immediately, before the browser processes any part of the XHTML markup. However, it's more common to use the <head> section for scripts that contain *functions* (see "Functions" below). Functions don't run immediately—instead, you summon them when your visitor takes some kind of action on a page, like moving a mouse.

Note: You can place as many <script> blocks in a web page as you want.

A typical script block consists of a series of programming instructions. To get a handle on how these instructions work, consider the following example, which makes your browser display a JavaScript alert box:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>JavaScript Test</title>
</head>
```

```
<body>
  <h1>You Will Be Wowed</h1>
  <p>This page uses JavaScript.</p>
  <script type="text/javascript">
    alert("Welcome, JavaScript coder.")
  </script>
</body>

</html>
```

This script pops up a window with a message, as shown in Figure 1-2. When you click OK, the message disappears, and it's back to life as usual for your web page.

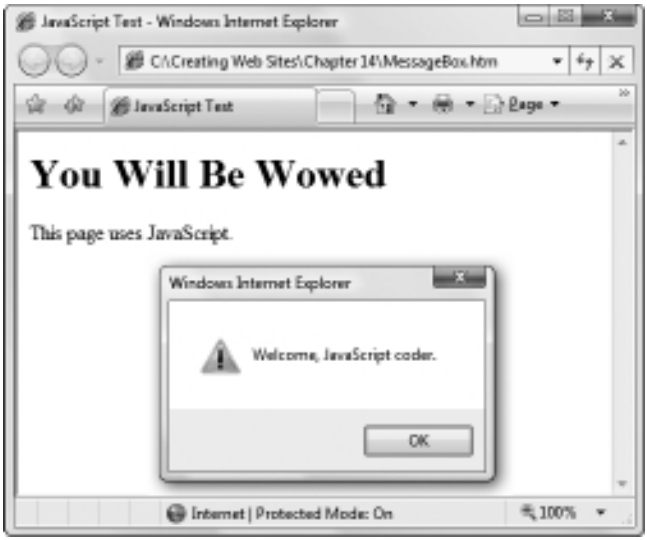

Figure 1-2: *Because you positioned the script element at the end of the XHTML markup for the page, the browser displays all the XHTML first and then pops up this alert box. If you put the script element at the beginning of the <body> section of your page (or in the <head> section), the alert box would appear earlier, while the page is still blank. The browser would then wait until you clicked OK before reading the rest of the XHTML page and displaying its contents.*

Like all scripts, the script in this example is wrapped inside a <script> element. The *type* attribute tells the browser that the script holds JavaScript code, which is an essential detail:

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

You're probably wondering exactly how this script works its magic. When a browser processes the script, it runs all the code, going one line at a time. In this case, there's only one line:

```
alert("Welcome, JavaScript coder.")
```

This line uses a built-in JavaScript function called an *alert*. A *function* is a programming routine consisting of one or more lines of code that performs a certain task. JavaScript has many built-in functions, but you can also build your own.

JavaScript's alert() function requires one piece of information, otherwise known as an *argument* in programmer-speak. In this case, that piece of information is the text you want displayed in the alert box. If you want to display an ordinary number, say 6, you could type it in as is—that is, you don't need to put it in quotes. But with text, there's no way for a browser to tell where text starts and stops. To handle this problem in JavaScript, you put text inside single apostrophe quotes (') or double quotation marks (''), as in the example above.

> ***Note:*** In programmer-speak, a distinct piece of text used in a program is called a *string*. "The friendly fox," "a," and "Rumpelstiltskin" all qualify as strings.

That's it. All this simple script does is call JavaScript's alert() function. (Spend enough time around programmers and JavaScript fans, and you'll soon learn that "call" is the preferred way to describe the action of triggering a function.) The alert() function does the rest, popping up the correct-sized window and waiting for your visitor to click OK.

> ***Note:*** To write this script, you need to know that there's an alert() function ready for you to use—a fact you can find out on one of the many JavaScript tutorial sites.

Based on what you now know, you should be able to change this script in the following ways:

- Display a different message (by changing the argument).
- Display more than one message box, one after the other (by adding more lines in your <script> block).
- Display the message box before your browser displays the web page (by changing the position of the <script> block).

It's not much to keep you occupied, but it does show how easy it is to get started using and changing a simple script.

Scripts and XHTML

As you've seen, XHTML uses the <script> block to hold JavaScript code. But it's important to understand that <script> is just one more XHTML element, and like all XHTML elements, it needs to follow the rules. One of those rules limits the characters allowed inside the <script> block. XHTML forbids certain characters, like the infamous angle brackets, because they have a special meaning in the XHTML language.

Unfortunately, special characters *do* sometimes crop up in code. You could replace them with character entities (see *http://www.webmonkey.com/reference/Special_Characters* for more about these), but that makes it far more difficult for other people to read, understand, and edit your code. A better solution is to wrap the entire script in something called a *CDATA section*—a specialized region of your XHTML document that suspends the usual XHTML rules. Here's how:

```
<script type="text/javascript">
//<![CDATA[
  alert("Welcome, JavaScript coder.")
//]]>
</script>
```

This technique is a bit ugly, but it does the trick. It's best not to think about it too deeply and just get into the habit of wrapping any lengthy block of script code in a CDATA section. The one exception is script code you put in a separate file (see "External Script Files" below)—your browser won't interpret it as XHTML, so you don't need to wrap it in a CDATA section.

> ***Note:*** For simplicity, this chapter uses a CDATA section only when absolutely necessary—in other words, when the script contains special characters that XHTML ordinarily wouldn't allow.

Browsers that don't support JavaScript

Some browsers will recognize the <script> element but refuse to execute your code. This can happen if a browser doesn't support JavaScript, or if JavaScript has been switched off. To deal with this situation, you can use the <noscript> element, which lets you supply alternate XHTML content.

You place the <noscript> element immediately after the <script> element. Here's an example that displays a paragraph of text for browsers that lack JavaScript support:

```
<script type="text/javascript">
  alert("Welcome, JavaScript coder.")
</script>
<noscript>
  <p>Welcome, non-JavaScript-enabled browser.</p>
</noscript>
```

GEM IN THE ROUGH

Dealing with Internet Explorer's Paranoia

If you run the alert example in Firefox, 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 appear in a yellow bar at the top of the page. Until you click the yellow bar, and then choose "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. However, there's no need to worry. In fact, 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 remove the security warning altogether, you can tell Internet Explorer to pretend you downloaded your web page from a web server. You do this by adding a special comment called the *Mark of the Web*. You place this comment immediately after the <html> element that begins your page:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<!-- saved from url=(0014)about:internet -->
```

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 XHTML comment.

UP TO SPEED

Spaces and Line Breaks in JavaScript

JavaScript code is quite tolerant of extra spaces. In this chapter, most of the examples use some sort of indenting to help you see the structure of the code. But as with XHTML, you don't absolutely have to add these spaces.

The only rule in JavaScript is that every code statement needs to be on a separate line. You can get around this limitation by using the line-termination character, which is a semicolon (;). For example, here's how you can compress three code statements onto one line:

```
alert("Hi"); alert("There"); alert("Dude");
```

Each semicolon designates the end of a code statement. This strange convention comes from the Bizarro world of C and Java.

If you don't want to put more than one code statement on the same line, you don't need the semicolons. However, you're still free to add them at the end of each line if you want. In fact, if you download a script from the Web, you just might find these optional semicolons, which is often a tip-off that a C or Java programmer wrote the script.

Variables

Every programming language includes the concept of *variables*, which are temporary containers that store important information. Variables can store numbers, objects, or pieces of text. As you'll see throughout this chapter, variables play a key role in many scripts, and they're a powerful tool in any programmer's arsenal.

Declaring variables

To create a variable in JavaScript, you use the *var* keyword, followed by the name of the variable. You can choose any name that makes sense to you, as long as you're consistent (and avoid spaces or special characters). Here's an example that creates a variable named *myMessage*:

```
var myMessage
```

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 its left. Here's an example that puts some text into myMessage:

```
myMessage = "Everybody loves variables"
```

Remember, you need to use quotation marks whenever you've got a text string. In contrast, if you want to copy a *number* into a variable, you don't need quotation marks:

```
myNumber = 27.3
```

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

You'll often want to create a variable and fill it with useful content, all in the same step. JavaScript lets you do so if you put an equal sign immediately after the variable name when you declare it:

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

To make matters a little confusing, JavaScript lets you refer to variables you haven't yet declared. Doing so is considered extremely bad form and is likely to cause all sorts of problems. However, it's worth knowing that these undeclared variables are permissible, because they're the source of many an unexpected error.

Modifying variables

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 addition and subtraction, then multiplication and division). 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
```

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

An example with variables

Although you'd need to read a thick volume to learn everything there is to know about variables, you can pick up a lot from a simple example. The following script inserts the current date into a web page. The example numbers each line of code to make it easy to reference.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>JavaScript Test</title>
</head>

<body>
  <h1>What Day Is It?</h1>
  <p>This page uses JavaScript.</p>
  <p>
  <script type="text/javascript">
1             var currentDate = new Date()
2             var message = "The current date is: "
3             message = message + currentDate.toDate-
String()
4             document.write(message)
  </script>
  </p>
</body>
</html>
```

Here's what's happening, line by line:

1. **This line creates a new variable named** *currentDate***.** It fills the currentDate variable (see number 3 below) with a new Date object. You'll know JavaScript is creating an object when you see the keyword new. (You'll learn more about objects in "Dynamic XHTML", below; for now, it's enough to know that objects come with built-in functions that work more or less the same way as the functions you learned about earlier.)
2. **This line creates a new variable named** *message***, and fills it with the beginning of a sentence that announces the date.**
3. **This line adds some new text to the end of the message you created in line 2.** The new text comes from the currentDate object. The tricky part is understanding that the currentDate object comes with a built-in function called toDateString(), which converts the date information it gets from your PC into a piece of text suitable for display in a browser (see Figure 1-3). Once again, this is the kind of detail you can only pick up by studying a good JavaScript reference.
4. **This line uses JavaScript's** *document* **object, which has a function named** *write()***.** The write() function copies a piece of text onto a web page at the current location. The final result is a page that shows your welcome message (see Figure 1-4).

Figure 1-3: *Some web page editors help you out when you write JavaScript code. For example, Expression Web displays a drop-down menu that shows you all the functions an object provides. Although this probably isn't enough for you to figure out how to use the Date object the first time out, it's a great way to refresh your memory later on.*

Scripts can get much more complex than this. For example, they can use loops to repeat a single action several times, or make decisions using conditional logic. You'll see examples of some of these techniques in this chapter, but you won't get a blow-by-blow exploration of the JavaScript language—in fact, that would require a small book of its own. If you want to learn more, check out the "Functions" section below.

Figure 1-4: *The document.write() command inserts text directly into a page, wherever you position the script block. In this case, the command displays the current date.*

POWER USERS' CLINIC

Becoming a JavaScript Guru

JavaScript requires some basic programming skills. However, it's fairly forgiving. Even non-geeks can learn to use JavaScript to create their own wildly customized programs (with the right motivation).

If you decide that you're not satisfied using other people's scripts and you want to create your own, it's time to learn more. Although in-depth JavaScript programming is beyond the scope of this book, there are plenty of great resources to get you started.

If you're happy learning from the Web, there's no shortage of tutorials. Three great places to start are *http://www.w3schools.com/js/*, *http://www.echoecho.com/javascript.htm/*, and *http://www.htmlgoodies.com/primers/jsp/*. If you prefer to have your advice printed and neatly bound, check out *JavaScript: The Missing Manual* (O'Reilly).

Functions

So far, you've seen simple scripts that use only a few lines of code. More realistic JavaScript scripts can run to dozens of lines and if you're not careful, they can grow into a grotesque tangle that leaves the rest of your page difficult to edit. To control the chaos, smart JavaScripters almost always use *custom functions*.

A function is a series of code instructions you group together and give a name. In a way, functions are like miniature programs, because they can perform a series of operations. The neat thing about functions is that you only need to create them once, and you can reuse them over and over again.

Declaring a function

To create a JavaScript function, start by deciding what your function is going to do (like display an alert message), and then choose a suitable name for it (like *ShowAlertBox*). As with most things in the programming world, function names can't have any spaces or special characters.

Armed with this information, you're ready to put a <script> block in the <head> section of your page. But this <script> block looks a little different from the examples you've seen so far. Here's a complete function that shows an alert box with a predefined message:

```
<script type="text/javascript">
  function ShowAlertBox() {
  alert("I'm a function.")
  }
</script>
```

To understand what's going on here, it helps to break down this example and consider it piece by piece.

Every function declaration starts with the word *function*, which tells JavaScript what you're up to.

```
function
```

Then, you want to name your function and add two parentheses. You use the parentheses to send extra information to your function, as you'll see in "Functions that receive information", below.

```
function ShowAlertBox()
```

At this point, you've finished *declaring* the function. All that remains is to add the code inside the function that actually makes it work. To do this, you need the funny curly braces shown in the alert box function above. The { brace indicates the start of your function code and the } brace indicates the end of it. You can put as many lines of code as you want in between.

One tricky part of function writing is the fact that JavaScript sets notoriously loose standards for line breaks. That means you can create a JavaScript function equivalent to the one above and put the curly braces on their own line, like this:

```
<script type="text/javascript">
  function ShowAlertBox()
  {
    alert("I'm a function.")
  }
</script>
```

But don't worry—both functions work exactly the same way.

> ***Tip:*** You can put as many functions as you want in a single <script> block. Just add them one after the other.

Calling a function

Creating a function is only half the battle. On their own, functions don't do anything. You have to *call* the function somewhere in your page to actually run the code. To call a function, you use the function name, followed by parentheses:

```
ShowAlertBox()
```

> ***Note:*** Don't leave out the parentheses after the function name. Otherwise, browsers will assume you're trying to use a variable rather than call a function.

You can call ShowAlertBox() anywhere you'd write ordinary JavaScript code. For example, here's a script that shows the alert message three times in a row to really hassle your visitors:

```
<script type="text/javascript">
  ShowAlert()
  ShowAlert()
  ShowAlert()
</script>
```

This is the same technique that, earlier, you saw used to call the alert() function. The difference is that alert() is built into JavaScript, while ShowAlertBox() is something you created yourself. Also, the alert() function requires one argument, while ShowAlertBox() doesn't use any.

Functions that receive information

The ShowAlertBox() function is beautifully simple. You simply call it, and it displays an alert box with the built-in message. Most functions don't work this easily. That's because 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 display a welcome message with some standard information in it, like the current date. But you also want the flexibility to change part of the message by substituting your own witty words each time you call the function. To do so, you need a way to call a function *and* to supply a text string with your message in it.

To solve this problem, you can create a ShowAlertBox() function that accepts a single argument. This argument represents the customized text you want to incorporate into your greeting. Choose a name for your customized text, say *customMessage*, and put it between the parentheses after the function name, like so:

```
function ShowAlertBox(customMessage) {
    ...
}
```

There's no limit to how many pieces of information a function can accept. You just need to separate each argument with a comma. Here's an example of the ShowAlertBox() function with three arguments named messageLine1, messageLine2, and messageLine3:

```
function ShowAlertBox(messageLine1, messageLine2,
messageLine3) {
    ...
}
```

Here's another example that shows a finished ShowAlertBox() function. It accepts a single argument named customMessage, and uses the customMessage argument to generate the text it displays in the alert box:

```
<script type="text/javascript">
1    function ShowAlertBox(customMessage)
2    {
3        // Get the date.
4        var currentDate = new Date()
5
6        // Build the full message.
7        var fullMessage = "** IMPORTANT BULLETIN **\n\n"
8        fullMessage += customMessage + "\n\n"
9        fullMessage += "Generated at: " + currentDate
         toTimeString() + "\n"
10       fullMessage += "This message courtesy of Magic-
         Media Inc."
11
12       // Show the message.
13       alert(fullMessage)
14   }
</script>
```

Here are some helpful notes to help you wade through the code:

- Any line that starts with // is a comment (see lines 3 and 6). Good programmers include lots of comments to help others understand how a function works (and help themselves remember what they did during a late-night coding binge). The browser ignores them.
- To put line breaks into an alert box, use the code *\n* (lines 7, 8, and 9). Each \n is equivalent to one line break. (This rule is for message boxes only. When writing XHTML, you add the familiar
 element to create a line break.)
- To build the text for the fullMessage variable (lines 7 to 10), the code uses a shortcut in the form of the += operator. This operator automatically takes whatever's on the *right* side of the equal sign and pastes it onto the end of the variable on the *left* side. In other words, this…

```
8        fullMessage += customMessage + "\n\n"
```

  … is equivalent to this longer line:

```
8        fullMessage = fullMessage + customMessage +
"\n\n"
```

Using this function is easy. You just need to remember that when you call the function, you have to supply the same number of arguments as you defined for the function, separating each one with a comma. In the case of the ShowAlertBox() function above, you only need to supply a single value for the customMessage variable. Here's an example:

```
<script type="text/javascript">
  ShowAlertBox("This web page includes JavaScript
functions.")
</script>
```

Figure 1-5 shows the result of this script.

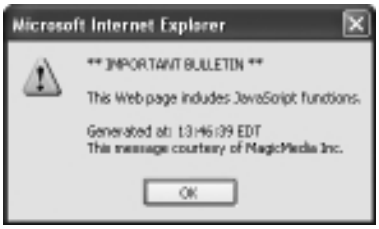

Figure 1-5: *This message is built out of several pieces of text, one of which you supplied as an argument to the ShowAlertBox() function.*

Functions that return information

Arguments let you send information *to* a function. You can also create functions that send information *back* to the script code that called them. The trick 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 you want your function to generate.

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

```
<script type="text/javascript">
  function MultiplyNumbers(numberA, numberB)
  {
    return numberA * numberB
  }
</script>
```

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

```
<p>The product of 3202 and 23405 is
<script type="text/javascript">
  var product = MultiplyNumbers(3202, 23405)
  document.write(product)
```

```
</script>
</p>
```

This XHTML shows a single line of text, followed by a block of script code. The script code calls the MultiplyNumbers() function, gets the result (the number 74942810), and stuffs it in a variable named *product* for later use. The code then uses the document.write() command to display the contents of the product variable on the page. The final result is a paragraph with this text:

```
The product of 3202 and 23405 is 74942810
```

To use a typical script you get from the Web, you need to copy one or more functions into your page. These functions are likely to look a lot more complex than what you've seen so far. However, now that you understand the basic structure of a function, you'll be able to wade through the code to get a fundamental understanding of what's taking place (or to at least pinpoint where the action is going down).

External Script Files

Reusing scripts inside a web page is neat, but did you know that you can share scripts *between* individual pages and even among different web sites? The trick is to put your script into an external file and then link to it.

For example, imagine you perfect the ShowAlertBox() routine so that it performs a complex task exactly the way you want it to, but it requires a couple of dozen lines of code to do so. To simplify your life and your XHTML document, you create a new file to store that script.

Script files are always plain text files. Usually, they have the extension *.js* (for JavaScript). You put all your code inside a script file, but

you don't include the <script> element. For example, you could create this JavaScript file named *ShowAlert.js*:

```
function ShowAlertBox()
{
  alert("This function is in an external file.")
}
```

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:

```
<script type="text/javascript" src="ShowAlert.js">
</script>
```

When a browser comes across this script block, it requests the ShowAlert.js file and treats it as though the code were right inside the page. Here's a complete XHTML test page that uses the ShowAlert.js file. The script in the body of the page calls the ShowAlertBox() function:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>Show Alert</title>
  <!-- Make all the functions in the ShowAlert.js file
    available in this page. Notice there's no actual con-
tent here. -->
  <script type="text/javascript" src="ShowAlert.js">
  </script>
</head>

<body>
  <!-- Test out one of the functions. -->
  <script type="text/javascript">
    ShowAlertBox()
  </script>
</body>

</html>
```

There's no difference in the way an embedded or external script works. However, storing your scripts in separate files helps keep your web site organized and makes it easy to reuse scripts across several pages. In fact, you can even link to JavaScript functions on another web site—just remember that the src attribute in the <script> block needs to point to a full URL (like *http://SuperScriptSite.com/ShowAlert.js/*) instead of just a file name. Of course, this technique is risky because the web site owner might rename, move, or modify the JavaScript file. If you really want to use the code, it's far better to copy it to your own web server so that that can't happen.

Note: Using separate script files doesn't improve your security one iota. Because anyone can request your script file, a savvy web visitor can figure out what scripts your page uses and take a look at them. So never include any code or secret details in a script that you don't want the world to know about.

Dynamic XHTML

JavaScript underwent a minor revolution in the late 1990s, adding support for a set of features called *Dynamic XHTML* (also referred to as Dynamic HTML, or shortened to DHTML). Dynamic XHTML isn't a new technology—instead, it's a fusion of three distinct ingredients:

- Scripting languages like JavaScript, which let you write code.
- The CSS (Cascading Style Sheet) standard, which lets you control how to position an XHTML element and how it appears.
- The XHTML *document object model* (or DOM), which lets you treat an XHTML page as a collection of *objects*.

The last point is the most important. Dynamic XHTML extends scripting languages like JavaScript so they can interact with a page as a collection of *objects*. This is a radical shift in web programming. Dynamic XHTML treats each XHTML element, including images, links, and even the lowly paragraph, as a separate programming ingredient that your JavaScript code can play with. Using these objects, you can change what each element looks like or even where your browser places them on a page.

XHTML Objects

Clearly, Dynamic XHTML requires a whole new way of thinking about web page design. Your scripts no longer look at your web page as a static block of XHTML. Instead, they see a combination of *objects*.

Before you can manipulate an object on your web page, you need a way to uniquely identify it. That way, your code can find the object it needs. The best choice for identifying objects is the *id* attribute. Add this attribute to the start tag for the element you want to manipulate, and choose a unique name, as shown here:

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

UP TO SPEED

Understanding Objects

In many programming languages, including JavaScript, everything revolves around objects. So what, exactly, is an object?

In the programming world, an object is nothing more than a convenient way to group some related features or information. For example, say you want to change the picture shown in an <img> element on a web page (which is useful if you want to write a script that flashes a series of images). The easiest way to interact with an <img> element in JavaScript is to use the corresponding *image* object. In effect, the image object is a container holding all sorts of potentially useful information about what's happening inside an <img> element (including its dimensions, its position, the name of the image file associated with it, and so on). The image object also gives you a way to manipulate the <img> element—that is, to change some or all of these details.

For example, you can use an image object to get information about the image, like this:

```
document.write("The tooltip says" + image.title)
```

You can even change one of these details. For example, you can modify the actual image that an <img> element shows by using this code:

```
image.src = "newpic.jpg"
```

You'll know an object's at work by the presence of a dot (.) in your code line. The dot separates the name of the variable (the first part) from one of the built-in functions it provides (called *methods*), or one of the related variables (called *properties*). You always put methods and properties after a period.

In the previous examples, *src* and *title* are two of the image object's properties. In other words, the code *image.src = "newpic.jpg"* is the equivalent of saying "Hey, Mr. Object named Image: I have a new picture for you. Change your src to point to *newpic.jpg*."

Programmers embraced objects long ago because they're a great way to organize code conceptually (not to mention a great way to share and reuse it). You might not realize it at first, but working with the image object is actually easier than memorizing a few dozen different commands that manipulate an image.

Note: In the example in "Modifying variables", you saw the document object working on a different task—displaying information in a web page. To accomplish this feat, the script used the write() method of the document object.

When you call the document.getElementById() method, you supply the ID of the XHTML element you're looking for. Here's an example that digs up the object for an XHTML 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 (titleObject), 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 XHTML 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 XHTML elements, like changing their style details 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"
```

If you run this code in a script, the header's text changes as soon as your browser runs this script.

The trick that makes this script work is the innerHTML *property*, which sets the content that's nested inside an element (in this case, the <title> element). Like all properties, innerHTML is just one aspect of an XHTML object you can alter. To write code statements like this, you need to know what properties are available for you to play with. Obviously, some properties apply to specific XHTML elements only, like the *src* attribute of an image. But modern browsers boast a huge catalog of DOM properties you can use with just about any XHTML element. Figure 1-13 lists some of the most useful.

> ***Tip:*** To get the properties that a specific XHTML element supports, check out the reference at *http://www.w3schools.com/htmldom/dom_reference.asp/*.

Currently, the above example works in two steps (getting the object, and then manipulating it). Although this two-step maneuver is probably the clearest approach, it's possible to combine these two steps into one line, which scripts often do. Here's an example:

```
document.getElementById("PageTitle").innerHTML = "This
Page Is Dynamic"
```

Remember, the advantage to getting an object first is that you can change several properties one after the other, without needing to look up the XHTML object using getElementById() each time.

Table 1-1. *Common XHTML object properties*

Property	Description
className	Lets you retrieve or set the class attribute. In other words, this property determines what style 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. innerHTML is insanely useful, but it has two quirks. First, it allows all XHTML content, including text and tags. So if you want to put bold text inside a paragraph, you can set innerHTML to Hi. Special characters aren't welcome—you need to replace them with character entities. 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 these items disappear, to be replaced by your new content. If you want to modify a specific piece of a paragraph, wrap that piece in a element.
parentElement	Provides the XHTML object for the element that contains this element. For example, if the current element is a element in a paragraph, parentElement gets the object for the <p> element. Once you have this object, you can modify the paragraph. Using this technique (and other similar techniques in Dynamic XHTML), you can jump from one element to another.
style	Bundles together all the CSS attributes that determine the appearance of the XHTML 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 myObject.style.fontSize. You can use the style object to set colors, borders, fonts, and even positioning.
tagName	Provides the name of the XHTML element for this object, without the angle brackets. For example, if the current object represents an <img> element, this returns the text "img".

Using XHTML objects in a script

The easiest way to come to grips with how XHTML objects work is to look at an example. The web pages shown in Figure 1-6 include a paragraph that continuously grows and then shrinks, as your code periodically tweaks the font size.

Figure 1-6: *If you were looking at this heading in a live web browser, you'd see that the text is always changing size, making it difficult to ignore.*

The way this example works is quite interesting. First of all, you define two variables in the <head> section of the underlying XHTML document. The *size* variable keeps track of the current size of the

text (which starts at 10 pixels). The *growIncrement* variable determines how much the text size changes each time your browser runs the code (initially, it grows by two pixels at a time):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Dynamic XHTML</title>
  <script type="text/javascript”>
//<![CDATA[
    // The current font size.
    var size = 10
    // The amount the font size is changing.
    var growIncrement = 2
```

Note: This script example is wrapped in a CDATA block because it uses the angle brackets to perform a numeric comparison.

Next, the script defines a function named ChangeFont(). This function retrieves the XHTML object, here the <p> element holding the text that will grow and shrink. Once again, the getElementById() function does the job:

```
function ChangeFont() {
  // Find object that represents the paragraph
  // whose text size you want to change.
  var paragraph = document.getElementById("animatedParagr
aph")
```

Now, using the size and growIncrement variables, you define a variable that performs a calculation to determine the new size for the paragraph:

```
size = size + growIncrement
```

In this example, the + performs a numeric addition, because both the size and growIncrement variables store a number.

It's just as easy to set the new size using the paragraph.style.fontSize property. Just tack the letters *px* on the end to indicate that your style setting is measured in pixels:

```
paragraph.style.fontSize = size + "px"
```

If this code runs perpetually, you'll eventually end up with text so ridiculously huge you can't see any of it on the page. To prevent this from happening, you add a safety valve to the code.

Say you decide that, when the text size hits 100, you want to stop enlarging it and start shrinking it. To do this, you write the script so that it sets the growIncrement variable to -2 when the text size reaches 100. This way, the text will start shrinking from this point on, two pixels at a time. To detect when the message has grown too big, you use conditional logic courtesy of the *if* statement. Here's what it looks like:

```
// Decide whether to reverse direction from
// growing to shrinking (or vice versa).
if (size > 100) {
  paragraph.innerHTML = "This Text is Shrinking"
  growIncrement = -2
}
```

Of course, you don't want the shrinking to go on forever, either. So it makes sense to add one last check that determines whether the text has shrunk to 10 pixels or less, in which case the script goes back to enlarging the text by setting growIncrement back to 2:

```
if (size < 10) {
  paragraph.innerHTML = "This Text is Growing"
  growIncrement = 2
}
```

Now, here comes the really crafty bit. JavaScript includes a setTimeout() function, which lets you give the browser an instruction that says "call this function, but wait a bit before you do." The setTimeout() function is handy when you want to create an interactive page. In this example, the setTimeout() function instructs the browser to call the ChangeFont() method again in 100 milliseconds (0.10 seconds):

```
    setTimeout("ChangeFont()", 100)
  }
//]]>
  </script>
</head>
```

Because the ChangeFont() function always uses setTimeout() to call itself again, the shrinking and resizing never stops. However, you can alter this behavior. You could, for example, add conditional logic so that JavaScript calls the setTimeout() method only a certain number of times.

The last detail is the <body> section, which contains the actual paragraph that you resize and a script that calls ChangeFont() for the first time, starting the whole process off:

```
<body>
  <p id="animatedParagraph">This Text is Growing</p>
  <script type="text/javascript">
     ChangeFont()
  </script>
</body>

</html>
```

Although the resizing paragraph trick is absurdly impractical, the technique is the basis for many much more impressive scripts (to get the whole script and play around with it, download it from the Missing CD page at *http://www.missingmanuals.com/*). For example, you can easily find scripts that animate text in various ways, making it fly in from the side of the page (see *http://www.codejunction.com/detailed/sequential-fly-in-text-effect.html/*); showing words appear one letter at a time, typewriter-style (*http://www.javascriptpage.com/tickert.html/*); or making a sparkle float over a title (*http://www.flooble.com/scripts/animate.php/*). Each of these examples uses the same basic approach, but adds significantly more code and gives you a much slicker effect.

Events

The most exciting JavaScript-powered pages are *dynamic*, which means they perform various actions as your visitor interacts with them (moving his mouse, typing in text, clicking on things, and so on). A dynamic page is far more exciting than an ordinary XHTML page, which appears in a browser in one shot and sits there, immobile.

To make dynamic pages, you program them to react to JavaScript *events*. Events are notifications that an XHTML element sends out when specific things happen.

For example, JavaScript gives every <a> hyperlink 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 XHTML element like a paragraph, link, image, table cell, or text box. That action triggers the onmouseover event and your code flies into action.

Here's an example that displays an alert message when a visitor moves his mouse pointer over a link:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

<head>
  <title>JavaScript Test</title>
  <meta http-equiv="Content-Script-Type" content="text/
javascript" />
</head>

<body>
  <h1>You Will Be Wowed (Again)</h1>
  <p>When you hover over <a href="SomePage.htm"
  onmouseover="alert('Colorless green ideas sleep furi-
ously.')">
```

```
this link</a>
  you'll see a secret message.
  </p>
</body>
</html>
```

When you write code to react to an event, you don't absolutely need a script block (although it's a good idea to use one anyway, as shown in the next section). Instead, you can just put your code between quotation marks next to the event attribute:

```
<a onmouseover="[Code goes here]">...</a>
```

There's one detail to keep in mind. In this example, the text argument ('Colorless green…') uses single quotation marks instead of double quotes. That's because the event attribute itself uses double quotes, and using double quotes for two different purposes at the same time will horribly confuse your browser.

When you attach code to an event, your browser assumes you're using JavaScript, which is by far the most popular scripting language. However, to meet the strictest rules of XHTML, you need to explicitly indicate your language choice by adding the following <meta> element to the <head> section of your page:

```
<meta http-equiv="Content-Script-Type" content="text/
javascript" />
```

Figure 1-7 shows the result of running this script and moving your mouse pointer over the link.

To use events effectively, you need to know what events JavaScript supports. In addition, you need to know which events work on which XHTML elements. Table 1-2 provides a list of commonly used events and the XHTML elements that they apply to (and you can find a more complete reference at *http://www.w3schools.com/html-dom/dom_reference.asp/*).

In the following sections, you'll see two common scenarios that use some of these events.

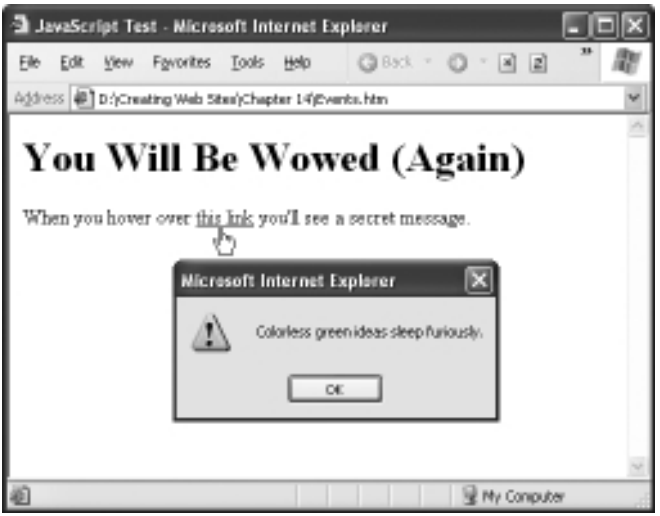

Figure 1-7: *In this example, the alert box doesn't pop up until you move your mouse pointer over the link.*

Note: XHTML requires that events use all-lowercase names, which is dreadfully unreadable. The event list in Table 1-2 breaks that rule so that you can review the event names more easily. However, when you use these events in your web pages, you need to replace the mixed-case name (like onMouseOver) with all lowercase letters (like onmouseover) to satisfy the laws of XHTML.

Table 1-2. *Common XHTML object properties*

Event Name (with uppercase letters)	Description	Applies To Which XHTML Elements
onClick	Triggered when you click an element.	Almost all
onMouseOver	Triggered when you move your mouse pointer over an element.	Almost all

onMouseOut	Triggered when you move your mouse pointer away from an element.	Almost all
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 the cursor appears there so you can type something in). Controls include text boxes, checkboxes, and so on.	<select>, <input>, <textarea>, <a>, <button>
onBlur	Triggered when focus leaves a control.	<select>, <input>, <textarea>, <a>, <button>
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 your browser fails to download an image (usually due to an incorrect URL).	<img>
onLoad	Triggered when your browser finishes downloading a new page or finishes loading an object, like an image.	<img>, <body>, <frame>, <frameset>
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>, <frameset>

Image Rollovers

One of the most popular mouse events is the *image rollover* event. To write one, you start by creating an <img> element that displays a specific picture. Then, when a visitor's mouse pointer moves over the image, your browser displays a new picture, thanks to the onmouseover event. Creating an image rollover is fairly easy. All you do is get the XHTML object for the <img> element, and then modify the *src* property.

In this situation, you can't get everything done with a single line of code. You could pile your entire script into the event attribute (using semicolons to separate each line), but that would be dreadfully confusing. A better choice is to write the code as a function. You can then hook the element up to the function using the event attribute.

For example, here's the function to swap an image. This example writes the function in a very generic way using parameters, so you can reuse it over and over, as you'll see in a moment. Every time you call the function, you indicate which image you want to change (by supplying the corresponding ID) and what new image file you want to use. That way, you can call the same function for any image rollover, anywhere on your page.

```
<script type="text/javascript">
  function ChangeImage(imageName, newImageFile) {
    // Find the object that represents the img element.
    var image = document.getElementById(imageName)

    // Change the picture.
    image.src = newImageFile
  }
</script>
```

When you create an image rollover, you need to use two events. The onmouseover event switches to the rollover picture, and the

onmouseout event (triggered when your visitor moves her mouse pointer *off* the XHTML element) switches back to the original picture.

```
<img id="SwappableImage" src="pic1.gif" alt=""
onmouseover="ChangeImage('SwappableImage', 'LostInterest-
Message.gif')"
onmouseout="ChangeImage('SwappableImage', 'ClickMe.gif')"
/>
```

Figure 1-8 shows the result.

Figure 1-8: *A rollover image in action.*

If you want to add more rollover images, just add a new <img> element with a different name. The following element uses the same initial image, but shows a different rollover image each time a visitor moves her mouse pointer on and off the image:

```
<img id="SwappableImage2" src="pic1.gif" alt=""
onmouseover="ChangeImage('SwappableImage2', 'MouseOver-
Picture.gif')"
onmouseout="ChangeImage('SwappableImage2', 'InitialPic-
ture.gif')" />
```

If you want to get really fancy, you can even use the onclick event (which occurs when you click an element) to throw yet another picture into the mix.

Collapsible Text

Another nifty way to use events is to create *collapsible pages*. The basic idea behind a collapsible page is this: If you've got a lot of information to show your viewers but don't want to overload them with a lengthy page, you hide (or collapse) chunks of text behind headlines that viewers can click when they want to see the details (see Figure 1-9).

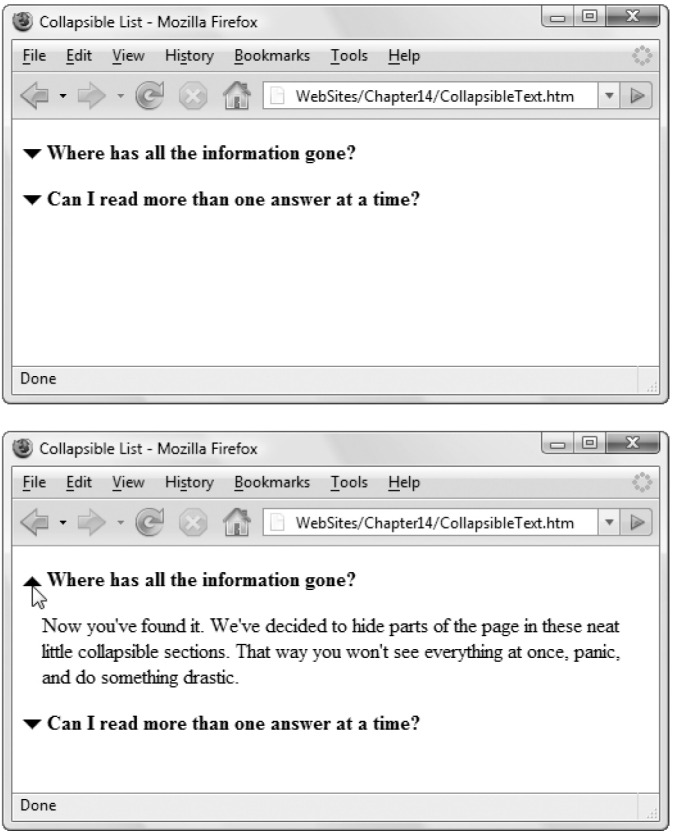

Figure 1-9: *Top: Initially, the browser hides all the body text.*

Bottom: But when you click the down arrow image, the browser displays the content in that section. You can reveal as many sections at a time as you want.

Dynamic XHTML gives you many ways to trick browsers into hiding text to create a collapsible page, and the next example shows one of the best. The technique revolves around the CSS *display* property. When you set this property to *block*, an item appears in the XHTML page in the normal way. But when you set it to *none*, the element disappears, along with everything inside it.

The first ingredient in making a collapsible page is to create the function that performs the hiding and the showing. The function requires two parameters: the name of the open/close image, and the name of the element you want to hide or show. The function actually does double duty. It checks the current state of the section, and then it changes that state. That means that it automatically shows a hidden section and automatically hides a displayed section, thanks to conditional logic. At the same time, the function changes the open/close image to display a different type of arrow.

Note: This practice, where you always reverse the current state of an item, is called *toggling* by jargon-happy programmers.

```
<script type="text/javascript">
  function ToggleVisibility(image, element){
    // Find the image.
    var image = document.getElementById(image)

    // Find the element to hide/unhide.
    var element = document.getElementById(element)

    // Check the element's current state.
    if (element.style.display == "none"){
      // If hidden, unhide it.
      element.style.display = "block"
      image.src = "open.png"
    }
    else
    {
      // If not hidden, hide it.
      element.style.display = "none"
      image.src = "closed.png"
```

```
    }
  }
</script>
```

The code starts out by looking up the two objects you need and storing them in the variables *image* and *element*. Then it gets to work. It looks at the current state of the paragraph and makes a decision (using an *if* statement) about whether it needs to show or hide the paragraph. Only one part of this conditional code runs. For example, if the browser is currently hiding the image (if the display style is *none*), the function runs just these two lines of code, then skips to the bottom of the function and ends:

```
element.style.display = "block"
image.src = "open.png"
```

On the other hand, if the browser is displaying the image, this code gets a chance to prove itself:

```
element.style.display = "none"
image.src = "closed.png"
```

To use this function, you need to add the <img> element that performs the toggling to your web page. You also need to add the XHTML section that contains the hidden content. You can show or hide virtually any XHTML element, but a good all-purpose choice is a element because you can stuff whatever you want to hide inside it. Here's an example:

```
<p>
  <img id="Question1Image" src="closed.png" alt=""
  onclick="ToggleVisibility('Question1Image','HiddenAnsw
er1')" />
  <b>Where has all the information gone?</b>
</p>

<div id="HiddenAnswer1">
  <p>Now you've found it. We've decided to hide parts of
  the page in these neat little collapsible sections.
  That way you won't see everything at once, panic, and
  do something drastic.</p>
</div>
```

The first part of the page markup (between the first <p> elements) defines the question heading, which visitors always see. It contains the image and the question (in bold). The second part (in the element) is the answer, which your code alternately shows or hides.

Best of all, because you put all the complicated stuff into a function, you can reuse the function to make additional collapsible sections. These sections have the same structure, but different contents:

```
<p>
  <img id="Question2Image" src="closed.png" alt=""
  onclick="ToggleVisibility('Question2Image','HiddenAnsw
er2')" />
  <b>Can I read more than one answer at a time?</b>
</p>

<div id="HiddenAnswer2" style="display:none">
  <p>You can expand as many or as few sections as you
  want. Once you've expanded a section, just click again
  to collapse it back up out of sight. The only rule is
  that when you leave this page and come back later,
  everthing will be hidden all over again. That's just
  the way JavaScript and Dynamic XHTML work.</p>
</div>
```

Notice that you have to give each <img> and element a unique id or your function won't know which picture to change and which section to hide.

Optionally, you can change this page to give it a different feel but keep the same collapsing behavior. For example, you can make the page easier to use by letting your visitor expand and collapse sections by clicking the heading text (instead of just the image). The easiest way to do this is to pop the image and the bold heading into a element, and then add the onclick event attribute to that element. Here's the change you need:

```
<div onclick="ToggleVisibility('Question1Image','HiddenAn
swer1')">
  <p>
  <img id="Question1Image" src="closed.png">
  <b>Where has all the information gone?</b>
  </p>
</div>
```

You could even underline the heading text so it looks like a link, which lets viewers know something will happen if they click it. Use the *text-decoration* style sheet property to underline the heading.

Finally, if you want all your collapsible sections to start off as collapsed, you need to add another script that performs this service. Here's the <script> block you need, which you can position at the end of your page, just before the closing </body> tag:

```
<script type="text/javascript">
  // Hide all sections, one by one.
  ToggleVisibility('Question1Image','HiddenAnswer1')
  ToggleVisibility('Question2Image','HiddenAnswer2')
  ...
</script>
```

You could hide your collapsible sections more easily by setting the *display* style property on each element with an inline style rule. However, this approach runs into trouble if a visitor turns off JavaScript, in which case every section will remain permanently hidden. By using the code approach shown here, you ensure that JavaScript-challenged browsers will simply display all the content, including the collapsible sections. The page won't be as impressive, but at least nothing goes missing.

An Interactive Form

You see some of the most powerful examples of JavaScript when you combine it with XHTML forms, which let you create graphical widgets like text boxes, checkboxes, buttons, and more. Without using a client-side programming language like JavaScript or a

more powerful server-side programming language, forms are quite limited. However, if you use JavaScript and add a dash of programming savvy, you can create pages that have their own intelligence.

For example, consider the page shown in Figure 1-10. It provides several text boxes where visitors type in numbers, and then it performs a calculation when visitors click a button.

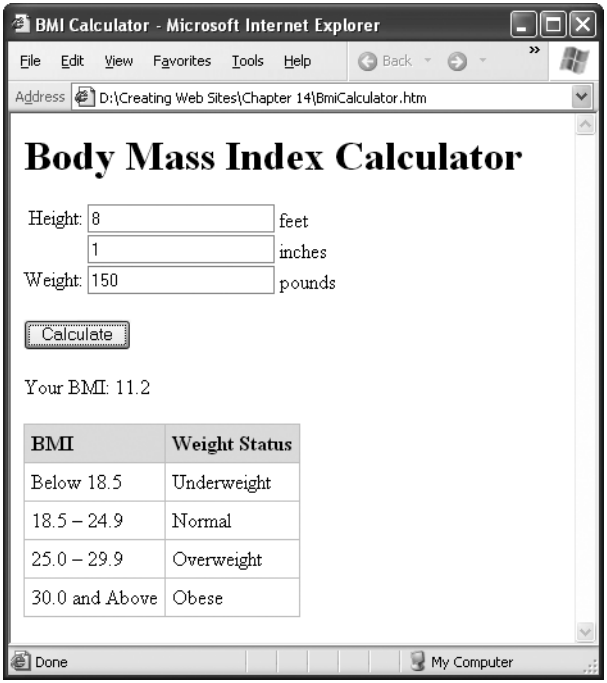

Figure 1-10: *BMI, or Body Mass Index, is a popular way to calculate a person's overall health by taking their height and weight into consideration. It produces a single number that you can compare against a few standard values. The BMI calculation is thought to be accurate for most people, but there are always exceptions.*

Building this example is surprisingly easy. The trickiest part is creating the function that powers the underlying calculations. This function needs several pieces of information, corresponding to the values in the three text boxes (feet, inches, and pounds). The function also needs the name of the element where it should display the results. Here's what the function looks like to start with:

```
<script type="text/javascript">
  function CalculateBMI(feet, inches, pounds, resultEle-
mentName) {
```

Tip: You could create a CalculateBMI() function that doesn't take any arguments. Instead, the function could just search for all the controls on the page by name. However, using arguments is always a good idea, because it makes your code more flexible. Now you can use the CalculateBMI() function on all kinds of different pages, with or without a form.

The function code that follows isn't much different from what you've seen before. One trick is that it begins by using a Number() function that's hardwired into JavaScript. This function converts the text a visitor types in to numbers that the function can use in its calculations. If you don't take this step, you might still get the right answer (sometimes), because JavaScript can automatically convert textual strings into numbers as needed. However, there's a catch—if you try to *add* two numbers and JavaScript thinks they're strings, it will just join the two strings into one piece of text, so 1+1 would get you 11. This mistake can really scramble your calculations, so it's best to always use the Number() function, like so:

```
inches = Number(inches)
pounds = Number(pounds)
feet = Number(feet)
```

The actual calculation isn't too interesting. It's taken straight from the definition of Body Mass Index, or BMI, which you can find on the Internet.

```
var totalInches = (feet * 12) + inches
```

Finally, the function displays the result on the page:

```
var resultElement = document.getElementById(resultElement
Name)
resultElement.innerHTML =
Math.round(pounds * 703 * 10 / totalInches / totalInches)
/ 10
}
</script>
```

Creating the form that uses this function is the easy part. All you do is create the text boxes with <input> elements and give them names you can easily remember. In this example, the form uses a table to make sure the text boxes line up neatly next to each other:

```
<form action="">
  <table>
    <tr>
    <td>Height: </td>
    <td><input type="text" name="feet" /> feet</td>
    </tr>
    <tr>
    <td> </td>
    <td><input type="text" name="inches" /> inches</td>
    </tr>
    <tr>
    <td>Weight: </td>
    <td><input type="text" name="pounds" /> pounds</td>
    </tr>
  </table>
```

Finally, at the bottom of the form, you create a button that calls the CalculateBMI() function using the form's values. To have the button make this call, you need to program your page to react to the onclick event. To look up a value in a form, you don't need the getElementById() function. Instead, you access it by name through the *this.form* object, which represents the current form:

```
    <p>
      <input type="button" name="calc" value="Calculate"
      onclick="CalculateBMI(this.form.feet.value, this.
form.inches.value,
this.form.pounds.value, 'result')" />
    </p>
</form>
```

The final ingredient is the element that displays the result. In this case, because you want it to appear inside another paragraph, the element makes more sense than the element.

```
<p>
  Your BMI: <span id="result"></span>
</p>
```

You can use all sorts of other form-related scripts. For example, you can check the information that people enter into forms for errors before letting them continue from one page to another. To learn more about these tricks, you need to take your search to the Web, as described in the next section.

Scripts on the Web

JavaScript is a truly powerful tool. If you're a die-hard alpha nerd who likes to program your TiVo to talk to your BlackBerry, you'll enjoy long nights of JavaScript coding. However, if you don't like to lie awake at night wondering what *var howMany = (trueTop>1?"s" : "");* really means, you'll probably be happier letting someone else do the heavy lifting.

If you fall into the nonprogrammer camp, this book has some very good news. The Web is flooded with free JavaScript. In fact, it's easier to find free scripts on the Web than free clip art, style sheets, or MIDI music. Most of the time, these scripts include step-by-step instructions that explain where to put the functions, what elements to use in your page, and how to hook your elements up to functions using events.

Although the list of JavaScript sites is too long to print, here are some good starting points:

- *http://webdeveloper.earthweb.com/webjs/*

  Offers a huge collection of JavaScript standards.

- *http://javascript.internet.com/*

  Provides a solid catalog of 2,000 bread-and-butter scripts.

- *http://www.javascript-2.com/*

  Tips the scales with a staggering 9,000 scripts.

- *http://www.dynamicdrive.com/*

  Provides a smaller set of scripts that emphasize modern programming techniques based on Dynamic XHTML. Includes exotic scripts like glowing green letters that tumble down the page, Matrix-style. Offers many scripts that are Internet Explorer-only, but clearly indicates browser support for each script.

- *http://www.javascripter.net/faq/*

  Unlike the other sites, this one doesn't offer a catalog of complete downloadable scripts. Instead, it's organized as a set of frequently asked JavaScript questions, with the relevant code for each answer.

- *http://www.webmonkey.com/tutorial/JavaScript_Tutorial/*

  Unlike the other sites, this one offers a smaller set of detailed JavaScript tutorials instead of a huge variety of standalone scripts. Useful if you want to learn more about some of the core JavaScript techniques.

Using this list, you can dig up everything from little frills to complete, functioning Tetris clones. But keep in mind that a script is only as good as the coder who created it. Even on sites with good quality control, you could stumble across a script that doesn't work on all browsers or slows your page down to a sluggish crawl. As a rule of thumb, always try out each script thoroughly before you start using it on your site.

UP TO SPEED

Script Categories

To get a handle on the types of Dynamic HTML scripts available, look through the different categories at Dynamic Drive (*http://www.dynamicdrive.com/*). Here's a sampling of what you'll find:

- The Calendars category has scripts that produce nifty XHTML that looks like calendars—great for displaying important dates or letting visitors plan in advance.
- The Date & Time category has live clocks and countdowns to a specific date.
- The Document Effects category has page transitions and background effects (like fireworks or floating stars).
- The Dynamic Content category has menus that slide out, sticky notes, and scrollable panels.
- The Form Effects category has scripts for managing forms. You can use them to make sure visitors submit forms only once, check for invalid entries, and more (see "An Interactive Form" above).
- The Games category has complete miniature games, like tic-tac-toe and Tetris. These games stretch the capabilities of JavaScript and Dynamic XHTML as far as they can go.
- The Image Effects category has slideshow and image gallery scripts, along with dynamic images that change pictures when you move your mouse.
- The Links & Tooltips category has fancy links that flash, button tricks, and pop-up text boxes that capture your visitors' attention.
- The Menus & Navigation category has handy collapsible menus and navigation bars that let visitors move through your site.
- The Mouse and Cursor category has scripts that change the mouse pointer and add those annoying mouse trails (pictures that follow the mouse pointer wherever it goes).
- The Scrollers category has marquee-style scrolling text, like you might see in a news ticker.
- The Text Animations category has scripts that bring text to life, making it shake, fly, glow, or take on even more bizarre characteristics.
- The User/System Preference category has scripts that dig up information about the browser that's currently displaying your page.
- The Window and Frames category has scripts for a dozen different types of pop-up windows.

Tip: The hallmark of a good script site is that it's easy to navigate. You'll know you've found a bad script site if it's so swamped with ads and pop-ups that you can't find the scripts themselves.

Finding a Cool Script

Ready to hunt for scripts online? The next series of steps takes you through the process from beginning to end.

1. **Fire up your browser and choose your site.**

   In this example, use *http://www.dynamicdrive.com/*.

2. **Choose the category you want from the site's home page (Figure 1-11).**

   In this case, use the Documents Effects category. For a sample of what else you can find, see the "Script Categories" box.

Figure 1-11: *The Dynamic Drive site organizes its scripts into clearly defined categories. If you're looking for something new, scroll down the page and you'll find links to the most recently added scripts. Some sites also provide quick links to reader favorites.*

3. **Scroll through the list of scripts in your category (Figure 1-12), and then click one.**

   In this case, use the Top-Down Stripy Curtain Script.

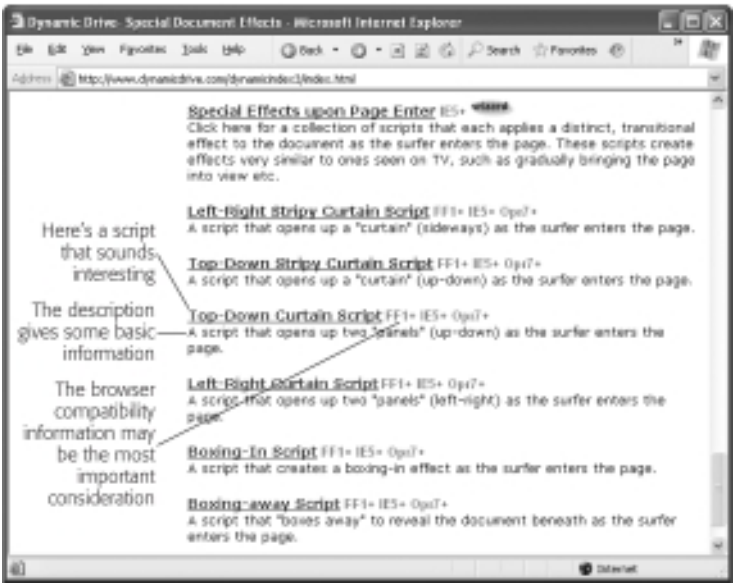

Figure 1-12: *The Top-Down Stripy Curtain Script is good to go, with support for Firefox, Internet Explorer 5 or greater, and Opera 7 or greater.*

4. **The next page shows an example of the script (Figure 1-13).**

   Once the page loads, you find a script description, the author's name, and a link to try the script out (if it isn't already being displayed on the page). Underneath all this information are the step-by-step instructions you need to use the script.

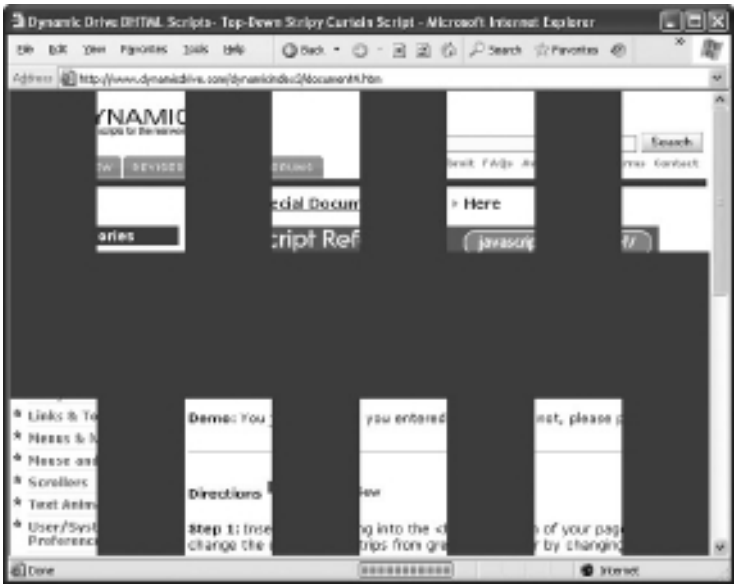

Figure 1-13: *Here's the Top-Down Stripy Curtain Script in action. It fills in the page by drawing alternating black strips, some from top to bottom and others from bottom to top. It all happens in a flash.*

5. **Follow the instructions to copy and paste the different parts of the script into your page (Figure 1-14).**

   Often, you get a set of functions you need to put in the <head> portion of your page and then some XHTML elements you need to place in the <body> section. In some cases, you can customize the scripts—for example, you might modify numbers and other values to tweak the script code, or change the XHTML elements to provide different content.

Note: Many scripts include a set of comments with author information. If they do, the rule usually is that you need to keep these comments in your script file, so other developers who check out your site will know where the code originally came from. This practice is just part of giving credit where credit's due. Ordinary web visitors won't even think to look at the script code, so they won't have any idea whether or not you wrote the script from scratch.

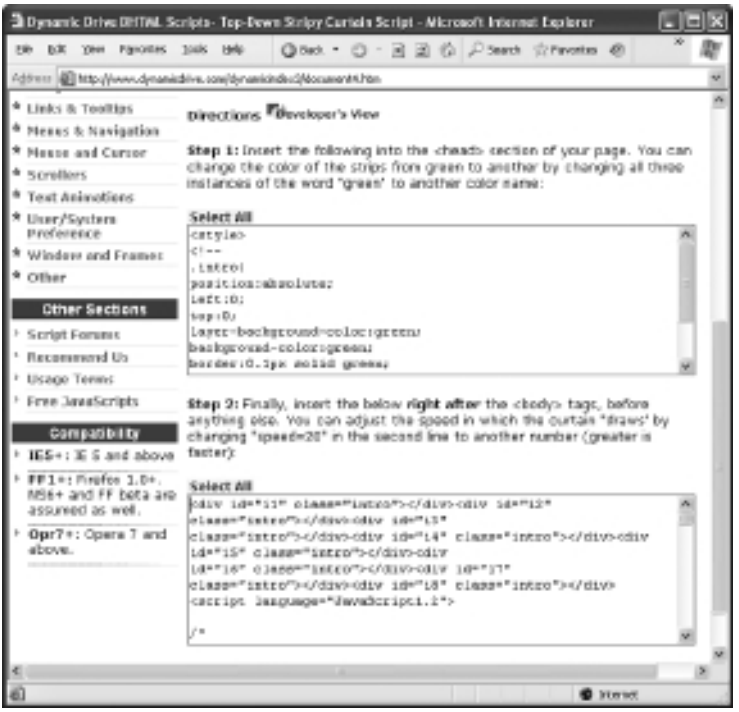

Figure 1-14: *The Top-Down Stripy Curtain Script has two components. The first part is a style definition that produces the solid background curtain that's wiped away with the page content. The second part creates the background curtain (as a element) and includes the script code that performs the transition. Copy both of these components to any page, and you're set. (For even better organization, consider placing the code in a separate JavaScript file, as described in the "External Script Files" section above.)*

Colophon

Peter McKie was the editor for *Add Interactivity to Your Site: The Mini Missing Manual*. Nellie McKesson was the production editor.

Nellie McKesson designed the interior layout, based on a series design by Phil Simpson. The text font for the PDF version of this book is Myriad Pro; and the heading and note font is Adobe Formata.

For best printing results of the PDF version of this book, use the following settings in the Adobe Reader Print dialog box: A: Pages: ii–[last page number]; B: Page Scaling: Mulitple pages per sheet; C: Pages per sheet: 2; D: Page Order: Horizontal.