Overview

JavaScript is a rich and expressive language in its own right. This section covers the basic concepts of JavaScript, as well as some frequent pitfalls for people who have not used JavaScript before. While it will be of particular value to people with no programming experience, even people who have used other programming languages may benefit from learning about some of the peculiarities of JavaScript.

If you're interested in learning more about the JavaScript language, I highly recommend *JavaScript: The Good Parts* by Douglas Crockford.

Syntax Basics

Understanding statements, variable naming, whitespace, and other basic JavaScript syntax.

A simple variable declaration

```
var foo = 'hello world';
```

Whitespace has no meaning outside of quotation marks

```
var foo =         'hello world';
```

Parentheses indicate precedence

```
2 * 3 + 5;    // returns 11; multiplication happens first
2 * (3 + 5);  // returns 16; addition happens first
```

Tabs enhance readability, but have no special meaning

```
var foo = function() {
    console.log('hello');
};
```

Operators

Basic Operators

Basic operators allow you to manipulate values.

Concatenation

```
var foo = 'hello';

var bar = 'world';

console.log(foo + ' ' + bar); // 'hello world'
```

Multiplication and division

```
2 * 3;
2 / 3;
```

Incrementing and decrementing

```
var i = 1;

var j = ++i;  // pre-increment:  j equals 2; i equals 2
var k = i++;  // post-increment: k equals 2; i equals 3
```

Operations on Numbers & Strings

In JavaScript, numbers and strings will occasionally behave in ways you might not expect.

Addition vs. concatenation

```
var foo = 1;
var bar = '2';

console.log(foo + bar);  // 12. uh oh
```

Forcing a string to act as a number

```
var foo = 1;
var bar = '2';

// coerce the string to a number
console.log(foo + Number(bar));
```

The Number constructor, when called as a function (like above) will have the effect of casting its argument into a number. You could also use the unary plus operator, which does the same thing:

Forcing a string to act as a number (using the unary-plus operator)

```
console.log(foo + +bar);
```

Logical Operators

Logical operators allow you to evaluate a series of operands using AND and OR operations.

Logical AND and OR operators

```
var foo = 1;
var bar = 0;
var baz = 2;

foo || bar;   // returns 1, which is true
bar || foo;   // returns 1, which is true

foo && bar;   // returns 0, which is false
foo && baz;   // returns 2, which is true
baz && foo;   // returns 1, which is true
```

Though it may not be clear from the example, the `||` operator returns the value of the first truthy operand, or, in cases where neither operand is truthy, it'll return the last of both operands. The `&&` operator returns the value of the first false operand, or the value of the last operand if both operands are truthy.

Be sure to consult the section called "Truthy and Falsy Things" for more details on which values evaluate to `true` and which evaluate to `false`.

Note

You'll sometimes see developers use these logical operators for flow control instead of using `if` statements. For example:

```
// do something with foo if foo is truthy
foo && doSomething(foo);

// set bar to baz if baz is truthy;
// otherwise, set it to the return
// value of createBar()
var bar = baz || createBar();
```

This style is quite elegant and pleasantly terse; that said, it can be really hard to read, especially for beginners. I bring it up here so you'll recognize it in code you read, but I don't recommend using it until you're extremely comfortable with what it means and how you can expect it to behave.

Comparison Operators

Comparison operators allow you to test whether values are equivalent or whether values are identical.

Comparison operators

```
var foo = 1;
var bar = 0;
var baz = '1';
var bim = 2;

foo == bar;   // returns false
foo != bar;   // returns true
foo == baz;   // returns true; careful!

foo === baz;             // returns false
foo !== baz;             // returns true
foo === parseInt(baz);   // returns true

foo > bim;    // returns false
bim > baz;    // returns true
foo <= baz;   // returns true
```

Conditional Code

Sometimes you only want to run a block of code under certain conditions. Flow control — via `if` and `else` blocks — lets you run code only under certain conditions.

Flow control

```
var foo = true;
var bar = false;

if (bar) {
    // this code will never run
    console.log('hello!');
}

if (bar) {
    // this code won't run
} else {
    if (foo) {
        // this code will run
    } else {
        // this code would run if foo and bar were both false
    }
}
```

Note

While curly braces aren't strictly required around single-line `if` statements, using them consistently, even when they aren't strictly required, makes for vastly more readable code.

Be mindful not to define functions with the same name multiple times within separate `if`/`else` blocks, as doing so may not have the expected result.

Conditional Variable Assignment with The Ternary Operator

Sometimes you want to set a variable to a value depending on some condition. You could use an `if/else` statement, but in many cases the ternary operator is more convenient. [Definition: The *ternary operator* tests a condition; if the condition is true, it returns a certain value, otherwise it returns a different value.]

The ternary operator

```
// set foo to 1 if bar is true;
// otherwise, set foo to 0
var foo = bar ? 1 : 0;
```

While the ternary operator can be used without assigning the return value to a variable, this is generally discouraged.

Switch Statements

Rather than using a series of if/else if/else blocks, sometimes it can be useful to use a switch statement instead. [Definition: *Switch statements* look at the value of a variable or expression, and run different blocks of code depending on the value.]

A switch statement

```
switch (foo) {

    case 'bar':
        alert('the value was bar -- yay!');
    break;

    case 'baz':
        alert('boo baz :(');
    break;

    default:
        alert('everything else is just ok');
    break;

}
```

Switch statements have somewhat fallen out of favor in JavaScript, because often the same behavior can be accomplished by creating an object that has more potential for reuse, testing, etc. For example:

```
var stuffToDo = {
    'bar' : function() {
        alert('the value was bar -- yay!');
    },

    'baz' : function() {
        alert('boo baz :(');
    },

    'default' : function() {
        alert('everything else is just ok');
    }
};

if (stuffToDo[foo]) {
    stuffToDo[foo]();
} else {
    stuffToDo['default']();
}
```

We'll look at objects in greater depth later in this chapter.

Loops

Loops let you run a block of code a certain number of times.

Loops

```
// logs 'try 0', 'try 1', ..., 'try 4'
for (var i=0; i<5; i++) {
    console.log('try ' + i);
}
```

Note that in Loops even though we use the keyword var before the variable name `i`*, this does not "scope" the variable* `i` *to the loop block. We'll discuss scope in depth later in this chapter.*

The for loop

A `for` loop is made up of four statements and has the following structure:

```
for ([initialisation]; [conditional]; [iteration])
 [loopBody]
```

The *initialisation* statement is executed only once, before the loop starts. It gives you an opportunity to prepare or declare any variables.

The *conditional* statement is executed before each iteration, and its return value decides whether or not the loop is to continue. If the conditional statement evaluates to a falsey value then the loop stops.

The *iteration* statement is executed at the end of each iteration and gives you an opportunity to change the state of important variables. Typically, this will involve incrementing or decrementing a counter and thus bringing the loop ever closer to its end.

The *loopBody* statement is what runs on every iteration. It can contain anything you want. You'll typically have multiple statements that need to be executed and so will wrap them in a block (`{...}`).

Here's a typical `for` loop:

A typical `for` loop

```
for (var i = 0, limit = 100; i < limit; i++) {
    // This block will be executed 100 times
    console.log('Currently at ' + i);
    // Note: the last log will be "Currently at 99"
}
```

The while loop

A `while` loop is similar to an `if` statement, except that its body will keep executing until the condition evaluates to false.

```
while ([conditional]) [loopBody]
```

Here's a typical `while` loop:

A typical `while` loop

```
var i = 0;
while (i < 100) {

    // This block will be executed 100 times
    console.log('Currently at ' + i);

    i++; // increment i

}
```

You'll notice that we're having to increment the counter within the loop's body. It is possible to combine the conditional and incrementer, like so:

A `while` loop with a combined conditional and incrementer

```
var i = -1;
while (++i < 100) {
    // This block will be executed 100 times
    console.log('Currently at ' + i);
}
```

Notice that we're starting at `-1` and using the prefix incrementer (`++i`).

The do-while loop

This is almost exactly the same as the `while` loop, except for the fact that the loop's body is executed at least once before the condition is tested.

```
do [loopBody] while ([conditional])
```

Here's a `do-while` loop:

A `do-while` loop

```
do {

    // Even though the condition evaluates to false
    // this loop's body will still execute once.

    alert('Hi there!');

} while (false);
```

These types of loops are quite rare since only few situations require a loop that blindly executes at least once. Regardless, it's good to be aware of it.

Breaking and continuing

Usually, a loop's termination will result from the conditional statement not evaluating to true, but it is possible to stop a loop in its tracks from within the loop's body with the `break` statement.

Stopping a loop

```
for (var i = 0; i < 10; i++) {
    if (something) {
        break;
    }
}
```

You may also want to continue the loop without executing more of the loop's body. This is done using the `continue` statement.

Skipping to the next iteration of a loop

```
for (var i = 0; i < 10; i++) {

    if (something) {
        continue;
    }

    // The following statement will only be executed
    // if the conditional 'something' has not been met
    console.log('I have been reached');

}
```

Reserved Words

JavaScript has a number of "reserved words," or words that have special meaning in the language. You should avoid using these words in your code except when using them with their intended meaning.

abstract	boolean	break	byte
case	catch	char	class
const	continue	debugger	default
delete	do	double	else
enum	export	extends	final
finally	float	for	function
goto	if	implements	import
in	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	super	switch
synchronized	this	throw	throws
transient	try	typeof	var
void	volatile	while	with

Arrays

Arrays are zero-indexed lists of values. They are a handy way to store a set of related items of the same type (such as strings), though in reality, an array can include multiple types of items, including other arrays.

A simple array

```
var myArray = [ 'hello', 'world' ];
```

Accessing array items by index

```
var myArray = [ 'hello', 'world', 'foo', 'bar' ];
console.log(myArray[3]);   // logs 'bar'
```

Testing the size of an array

```
var myArray = [ 'hello', 'world' ];
console.log(myArray.length);   // logs 2
```

Changing the value of an array item

```
var myArray = [ 'hello', 'world' ];
myArray[1] = 'changed';
```

While it's possible to change the value of an array item as shown in "Changing the value of an array item", it's generally not advised.

Adding elements to an array

```
var myArray = [ 'hello', 'world' ];
myArray.push('new');
```

Working with arrays

```
var myArray = [ 'h', 'e', 'l', 'l', 'o' ];
var myString = myArray.join('');   // 'hello'
var mySplit = myString.split('');  // [ 'h', 'e', 'l', 'l', 'o' ]
```

Objects

Objects contain one or more key-value pairs. The key portion can be any string. The value portion can be any type of value: a number, a string, an array, a function, or even another object.

[Definition: When one of these values is a function, it's called a *method* of the object.] Otherwise, they are called properties.

As it turns out, nearly everything in JavaScript is an object — arrays, functions, numbers, even strings — and they all have properties and methods.

Creating an "object literal"

```
var myObject = {
    sayHello : function() {
        console.log('hello');
    },
    myName : 'Rebecca'
};
myObject.sayHello();             // logs 'hello'
console.log(myObject.myName);    // logs 'Rebecca'
```

Note When creating object literals, you should note that the key portion of each key-value pair can be written as any valid JavaScript identifier, a string (wrapped in quotes) or a number:

```
var myObject = {
    validIdentifier: 123,
    'some string': 456,
    99999: 789
};
```

Object literals can be extremely useful for code organization; for more information, read Using Objects to Organize Your Code by Rebecca Murphey.

Functions

Functions contain blocks of code that need to be executed repeatedly. Functions can take zero or more arguments, and can optionally return a value.

Functions can be created in a variety of ways:

Function Declaration

```
function foo() { /* do something */ }
```

Named Function Expression

```
var foo = function() { /* do something */ }
```

I prefer the named function expression method of setting a function's name, for some rather in-depth and technical reasons. You are likely to see both methods used in others' JavaScript code.

Using Functions

A simple function

```
var greet = function(person, greeting) {
    var text = greeting + ', ' + person;
    console.log(text);
};
greet('Rebecca', 'Hello');
```

A function that returns a value

```
var greet = function(person, greeting) {
    var text = greeting + ', ' + person;
    return text;
};

console.log(greet('Rebecca','hello'));
```

A function that returns another function

```
var greet = function(person, greeting) {
    var text = greeting + ', ' + person;
    return function() { console.log(text); };
};


var greeting = greet('Rebecca', 'Hello');
greeting();
```

Self-Executing Anonymous Functions

A common pattern in JavaScript is the self-executing anonymous function. This pattern creates a function expression and then immediately executes the function. This pattern is extremely useful for cases where you want to avoid polluting the global namespace with your code — no variables declared inside of the function are visible outside of it.

A self-executing anonymous function

```
(function(){
    var foo = 'Hello world';
})();


console.log(foo);    // undefined!
```

Functions as Arguments

In JavaScript, functions are "first-class citizens" — they can be assigned to variables or passed to other functions as arguments. Passing functions as arguments is an extremely common idiom in jQuery.

Passing an anonymous function as an argument

```
var myFn = function(fn) {
    var result = fn();
    console.log(result);
};

myFn(function() { return 'hello world'; });    // logs 'hello world'
```

Passing a named function as an argument

```
var myFn = function(fn) {
    var result = fn();
    console.log(result);
};

var myOtherFn = function() {
    return 'hello world';
};

myFn(myOtherFn);    // logs 'hello world'
```

Testing Type

JavaScript offers a way to test the "type" of a variable. However, the result can be confusing — for example, the type of an Array is "object".

It's common practice to use the `typeof` operator when trying to determining the type of a specific value.

Testing the type of various variables

```
var myFunction = function() {
    console.log('hello');
};

var myObject = {
    foo : 'bar'
};

var myArray = [ 'a', 'b', 'c' ];
```

The `this` keyword

In JavaScript, as in most object-oriented programming languages, `this` is a special keyword that is used within methods to refer to the object on which a method is being invoked. The value of `this` is determined using a simple series of steps:

1. If the function is invoked using Function.call or Function.apply, `this` will be set to the first argument passed to call/apply. If the first argument passed to call/apply is `null` or `undefined`, `this` will refer to the global object (which is the `window` object in Web browsers).
2. If the function being invoked was created using Function.bind, `this` will be the first argument that was passed to bind at the time the function was created.
3. If the function is being invoked as a method of an object, `this` will refer to that object.
4. Otherwise, the function is being invoked as a standalone function not attached to any object, and `this` will refer to the global object.

A function invoked using Function.call

```
var myObject = {
    sayHello : function() {
        console.log('Hi! My name is ' + this.myName);
    },

    myName : 'Rebecca'
};

var secondObject = {
    myName : 'Colin'
};

myObject.sayHello();                  // logs 'Hi! My name is Rebecca'
myObject.sayHello.call(secondObject); // logs 'Hi! My name is Colin'
```

A function created using Function.bind

```
var myName = 'the global object',

    sayHello = function () {
        console.log('Hi! My name is ' + this.myName);
    },

    myObject = {
        myName : 'Rebecca'
    };

var myObjectHello = sayHello.bind(myObject);

sayHello();       // logs 'Hi! My name is the global object'
myObjectHello();  // logs 'Hi! My name is Rebecca'
```

A function being attached to an object at runtime

```
var myName = 'the global object',

    sayHello = function() {
        console.log('Hi! My name is ' + this.myName);
    },

    myObject = {
        myName : 'Rebecca'
    },

    secondObject = {
        myName : 'Colin'
    };

myObject.sayHello = sayHello;
secondObject.sayHello = sayHello;

sayHello();               // logs 'Hi! My name is the global object'
myObject.sayHello();      // logs 'Hi! My name is Rebecca'
secondObject.sayHello();  // logs 'Hi! My name is Colin'
```

Note

When invoking a function deep within a long namespace, it is often tempting to reduce the amount of code you need to type by storing a reference to the actual function as a single, shorter variable. It is important not to do this with instance methods as this will cause the value of `this` within the function to change, leading to incorrect code operation. For instance:

```
var myNamespace = {
    myObject : {
        sayHello : function() {
            console.log('Hi! My name is ' + this.myName);
        },

        myName : 'Rebecca'
    }
};

var hello = myNamespace.myObject.sayHello;

hello();  // logs 'Hi! My name is undefined'
```

You can, however, safely reduce everything up to the object on which the method is invoked:

```
var myNamespace = {
    myObject : {
        sayHello : function() {
            console.log('Hi! My name is ' + this.myName);
        },

        myName : 'Rebecca'
    }
};

var obj = myNamespace.myObject;

obj.sayHello();  // logs 'Hi! My name is Rebecca'
```

Scope

"Scope" refers to the variables that are available to a piece of code at a given time. A lack of understanding of scope can lead to frustrating debugging experiences.

When a variable is declared inside of a function using the `var` keyword, it is only available to code inside of that function — code outside of that function cannot access the variable. On the other hand, functions defined *inside* that function *will* have access to to the declared variable.

Furthermore, variables that are declared inside a function without the `var` keyword are not local to the function — JavaScript will traverse the scope chain all the way up to the window scope to find where the variable was previously defined. If the variable wasn't previously defined, it will be defined in the global scope, which can have extremely unexpected consequences;

Functions have access to variables defined in the same scope

```
var foo = 'hello';

var sayHello = function() {
    console.log(foo);
};

sayHello();         // logs 'hello'
console.log(foo);   // also logs 'hello'
```

Code outside the scope in which a variable was defined does not have access to the variable

```
var sayHello = function() {
    var foo = 'hello';
    console.log(foo);
};

sayHello();         // logs 'hello'
console.log(foo);   // doesn't log anything
```

Variables with the same name can exist in different scopes with different values

```
var foo = 'world';

var sayHello = function() {
    var foo = 'hello';
    console.log(foo);
};

sayHello();         // logs 'hello'
console.log(foo);   // logs 'world'
```

Functions can "see" changes in variable values after the function is defined

Introducing JavaScript

JavaScript was developed by Netscape Communications Corporation, the maker of the Netscape web browser. JavaScript was the first web scripting language to be supported by browsers, and it is still by far the most popular.

JavaScript is almost as easy to learn as HTML, and it can be included directly in HTML documents. Here are a few of the things you can do with JavaScript:

- Display messages to the user as part of a web page, in the browser's status line, or in alert boxes
- Validate the contents of a form and make calculations (for example, an order form can automatically display a running total as you enter item quantities)
- Animate images or create images that change when you move the mouse over them
- Create ad banners that interact with the user, rather than simply displaying a graphic

- Detect the browser in use or its features and perform advanced functions only on browsers that support them
- Detect installed plug-ins and notify the user if a plug-in is required
- Modify all or part of a web page without requiring the user to reload it
- Display or interact with data retrieved from a remote server

You can do all this and more with JavaScript, including creating entire applications. We'll explore the uses of JavaScript throughout this book.

How JavaScript Fits into a Web Page

Using the `<script>` tag, you can add a short script (in this case, just one line) to a web document, as shown in Listing 4.1. The `<script>` tag tells the browser to start treating the text as a script, and the closing `</script>` tag tells the browser to return to HTML mode. In most cases, you can't use JavaScript statements in an HTML document except within `<script>` tags. The exception is event handlers, described later in this chapter.

LISTING 4.1 A Simple HTML Document with a Simple Script

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>The American Eggplant Society</title>
  </head>

  <body>
    <h1>The American Eggplant Society</h1>
    <p>Welcome to our site. Unfortunately, it is still
       under construction.</p>
    <p>We last worked on it on this date:
    <script type="text/javascript">
    <!-- Hide the script from old browsers
    document.write(document.lastModified);
    // Stop hiding the script -->
    </script>
    </p>
  </body>
</html>
```

JavaScript's `document.write` statement, which you'll learn more about later, sends output as part of the web document. In this case, it displays the modification date of the document, as shown in Figure 4.1.

In this example, we placed the script within the body of the HTML document. There are actually four different places where you might use scripts:

- **In the body of the page**—In this case, the script's output is displayed as part of the HTML document when the browser loads the page.
- **In the header of the page between the `<head>` tags**—Scripts in the header can't create output within the HTML document, but can be referred to by other scripts. The header is often used for functions—groups of JavaScript statements that can be used as a single unit. You will learn more about functions in Chapter 14, "Getting Started with JavaScript Programming."
- **Within an HTML tag, such as `<body>` or `<form>`**—This is called an *event handler* and enables the script to work with HTML elements. When using JavaScript in event handlers, you don't need to use the `<script>` tag. You'll learn more about event handlers in Chapter 14.
- **In a separate file entirely**—JavaScript supports the use of files with the .js extension containing scripts; these can be included by specifying a file in the `<script>` tag.

Using Separate JavaScript Files

When you create more complicated scripts, you'll quickly find your HTML documents become large and confusing. To avoid this, you can use one or more external JavaScript files. These are files with the .js extension that contain JavaScript statements.

External scripts are supported by all modern browsers. To use an external script, you specify its filename in the `<script>` tag:

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

Because you'll be placing the JavaScript statements in a separate file, you don't need anything between the opening and closing `<script>` tags—in fact, anything between them will be ignored by the browser.

You can create the .js file using a text editor. It should contain one or more JavaScript commands and only JavaScript—don't include `<script>` tags, other HTML tags, or HTML comments. Save the .js file in the same directory as the HTML documents that refer to it.

TIP

External JavaScript files have a distinct advantage: You can link to the same .js file from two or more HTML documents. Because the browser stores this file in its cache, this can reduce the time it takes for your web pages to display.

Understanding JavaScript Events

Many of the useful things you can do with JavaScript involve interacting with the user and that means responding to *events*—for example, a link or a button being clicked. You can define event handlers within HTML tags to tell the browser how to respond to an event. For example, Listing 4.2 defines a button that displays a message when clicked.

LISTING 4.2 A Simple Event Handler

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Event Test</title>
  </head>

  <body>
    <h1>Event Test</h1>
    <button type="button"
            onclick="alert('You clicked the button.')">
            Click Me!</button>
  </body>
</html>
```

Exploring JavaScript's Capabilities

If you've spent any time browsing the Web, you've undoubtedly seen lots of examples of JavaScript in action. Here are some brief descriptions of typical applications for JavaScript, all of which you'll explore further, later in this book.

Improving Navigation

Some of the most common uses of JavaScript are in navigation systems for websites. You can use JavaScript to create a navigation tool—for example, a drop-down menu to select the next page to read or a submenu that pops up when you hover over a navigation link.

When it's done right, this kind of JavaScript interactivity can make a site easier to use, while remaining usable for browsers that don't support JavaScript.

Validating Forms

Form validation is another common use of JavaScript. A simple script can read values the user types into a form and can make sure they're in the right format, such as with ZIP Codes or phone numbers. This allows users to notice common errors and fix them without waiting for a response from the web server. You'll learn how to work with form data in Chapter 26, "Working with Web-Based Forms."

Special Effects

One of the earliest and most annoying uses of JavaScript was to create attention-getting special effects—for example, scrolling a message in the browser's status line or flashing the background color of a page.

These techniques have fortunately fallen out of style, but thanks to the W3C DOM and the latest browsers, some more impressive effects are possible with JavaScript—for example, creating objects that can be dragged and dropped on a page or creating fading transitions between images in a slideshow.

Remote Scripting (AJAX)

For a long time, the biggest limitation of JavaScript was that there was no way for it to communicate with a web server. For example, you could use it to verify that a phone number had the right number of digits, but not to look up the user's location in a database based on the number.

Now that some of JavaScript's advanced features are supported by most browsers, this is no longer the case. Your scripts can get data from a server without loading a page or send data back to be saved. These features are collectively known as AJAX (Asynchronous JavaScript And XML), or *remote scripting*. You'll learn how to develop AJAX scripts in Chapter 24, "AJAX: Remote Scripting."

You've seen AJAX in action if you've used Google's Gmail mail application or recent versions of Yahoo! Mail or Microsoft Hotmail. All of these use remote scripting to present you with a responsive user interface that works with a server in the background.

Displaying Time with JavaScript

One common and easy use for JavaScript is to display dates and times. Because JavaScript runs on the browser, the times it displays will be in the user's current time zone. However, you can also use JavaScript to calculate "universal" (UTC) time.

As a basic introduction to JavaScript, you will now create a simple script that displays the current time and the UTC time within a web page, starting with the next section.

NOTE

UTC stands for Universal Time Coordinated, and is the atomic time standard based on the old GMT (Greenwich Mean Time) standard. This is the time at the Prime Meridian, which runs through Greenwich, London, England.

Beginning the Script

Your script, like most JavaScript programs, begins with the HTML `<script>` tag. As you learned earlier in this chapter, you use the `<script>` and `</script>` tags to enclose a script within the HTML document.

To begin creating the script, open your favorite text editor and type the beginning and ending `<script>` tags as shown.

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

CAUTION

Remember to include only valid JavaScript statements between the starting and ending `<script>` tags. If the browser finds anything but valid JavaScript statements within the `<script>` tags, it will display a JavaScript error message.

Adding JavaScript Statements

Your script now needs to determine the local and UTC times, and then display them to the browser. Fortunately, all the hard parts, such as converting between date formats, are built in to the JavaScript interpreter.

Storing Data in Variables

To begin the script, you will use a *variable* to store the current date. You will learn more about variables in Chapter 16, "Using JavaScript Variables, Strings, and Arrays." A *variable* is a container that can hold a value—a number, some text, or in this case, a date.

To start writing the script, add the following line after the first `<script>` tag. Be sure to use the same combination of capital and lowercase letters in your version because JavaScript commands and variable names are case sensitive.

```
now = new Date();
```

This statement creates a variable called `now` and stores the current date and time in it. This statement and the others you will use in this script use JavaScript's built-in `Date` object, which enables you to conveniently handle dates and times. You'll learn more about working with dates in Chapter 17, "Using JavaScript Functions and Objects."

Calculating the Results

Internally, JavaScript stores dates as the number of milliseconds since January 1, 1970. Fortunately, JavaScript includes a number of functions to convert dates and times in various ways, so you don't have to figure out how to convert milliseconds to day, date, and time.

To continue your script, add the following two statements before the final `</script>` tag:

```
localtime = now.toString();
utctime = now.toGMTString();
```

These statements create two new variables: `localtime`, containing the current time and date in a nice readable format, and `utctime`, containing the UTC equivalent.

Creating Output

You now have two variables—`localtime` and `utctime`—which contain the results we want from our script. Of course, these variables don't do us much good unless we can see them. JavaScript includes a number of ways to display information, and one of the simplest is the `document.write` statement.

The `document.write` statement displays a text string, a number, or anything else you throw at it. Because your JavaScript program will be used within a web page, the output will be displayed as part of the page. To display the result, add these statements before the final `</script>` tag:

```
document.write("<strong>Local time:</strong> " + localtime + "<br/>");
document.write("<strong>UTC time:</strong> " + utctime);
```

These statements tell the browser to add some text to the web page containing your script. The output will include some brief strings introducing the results and the contents of the `localtime` and `utctime` variables.

Notice the HTML tags, such as `<strong>`, within the quotation marks—because JavaScript's output appears within a web page, it needs to be formatted using HTML. The `<br/>` tag in the first line ensures that the two times will be displayed on separate lines.

Adding the Script to a Web Page

You should now have a complete script that calculates a result and displays it. Your listing should match Listing 4.3.

LISTING 4.3 The Complete Date and Time Script

```
<script type="text/javascript">
now = new Date();
localtime = now.toString();
utctime = now.toGMTString();
document.write("<strong>Local time:</strong> " + localtime + "<br/>");
document.write("<strong>UTC time:</strong> " + utctime);
</script>
```

To use your script, you'll need to add it to an HTML document. If you use the general template you've seen in the chapters so far, you should end up with something like Listing 4.4.

LISTING 4.4 The Date and Time Script in an HTML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Displaying Times and Dates</title>
  </head>

  <body>
    <h1>Current Date and Time</h1>
    <script type="text/javascript">
     now = new Date();
     localtime = now.toString();
     utctime = now.toGMTString();
     document.write("<strong>Local time:</strong> "
          + localtime + "<br/>");
    document.write("<strong>UTC time:</strong> " + utctime);
   </script>
  </body>
</html>
```

Now that you have a complete HTML document, save it with the .htm or .html extension.

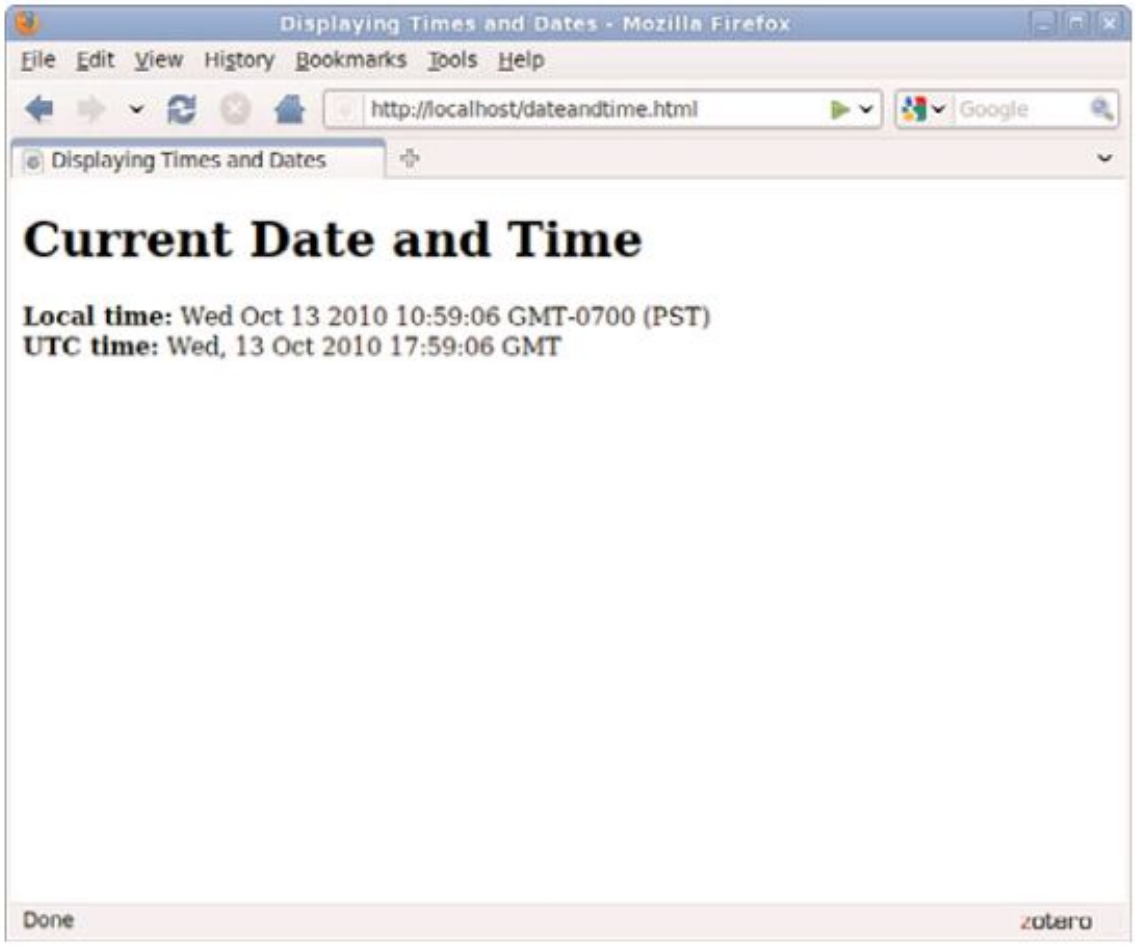
Testing the Script

To test your script, you simply need to load the HTML document you created in a web browser. If you typed the script correctly, your browser should display the result of the script, as shown in Figure 4.2. (Of course, your result won't be the same as mine, but it should be the same as the setting of your computer's clock.)

A note about Internet Explorer 6.0 and above: Depending on your security settings, the script might not execute, and a yellow highlighted bar on the top of the browser might display a security warning. In this case, click the yellow bar and select Allow Blocked Content to allow your script to run. (This happens because the default security settings allow JavaScript in online documents, but not in local files.)

Modifying the Script

Although the current script does indeed display the current date and time, its display isn't nearly as attractive as the clock on your wall or desk. To remedy that, you can use some additional JavaScript features and a bit of HTML to display a large clock.



To display a large clock, we need the hours, minutes, and seconds in separate variables. Once again, JavaScript has built-in functions to do most of the work:

```
hours = now.getHours();
mins = now.getMinutes();
secs = now.getSeconds();
```

These statements load the `hours`, `mins`, and `secs` variables with the components of the time using JavaScript's built-in date functions.

After the hours, minutes, and seconds are in separate variables, you can create `document.write` statements to display them:

```
document.write("<h1>");
document.write(hours + ":" + mins + ":" + secs);
document.write("</h1>");
```

The first statement displays an HTML `<h1>` header tag to display the clock in a large typeface. The second statement displays the `hours`, `mins`, and `secs` variables, separated by colons, and the third adds the closing `</h1>` tag.

You can add the preceding statements to the original date and time script to add the large clock display. Listing 4.5 shows the complete modified version of the script.

LISTING 4.5 The Date and Time Script with Large Clock Display

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Displaying Times and Dates</title>
  </head>

  <body>
    <h1>Current Date and Time</h1>
    <script type="text/javascript">
     now = new Date();
     localtime = now.toString();
     utctime = now.toGMTString();
     document.write("<strong>Local time:</strong> "
         + localtime + "<br/>");
    document.write("<strong>UTC time:</strong> " + utctime);
    hours = now.getHours();
    mins = now.getMinutes();
    secs = now.getSeconds();
    document.write("<h1>");
    document.write(hours + ":" + mins + ":" + secs);
    document.write("</h1>");
   </script>
  </body>
</html>
```

Now that you have modified the script, save the HTML file and open the modified file in your browser. If you left the browser running, you can simply use the Reload button to load the new version of the script. Try it and verify that the same time is displayed in both the upper portion of the window and the new large clock. Figure 4.3 shows the results.

Dealing with JavaScript Errors

As you develop more complex JavaScript applications, you're going to run into errors from time to time. JavaScript errors are usually caused by mistyped JavaScript statements.

To see an example of a JavaScript error message, modify the statement you added in the previous section. We'll use a common error: omitting one of the parentheses. Change the last `document.write` statement in Listing 4.5 to read:

```
document.write("</h1>";
```

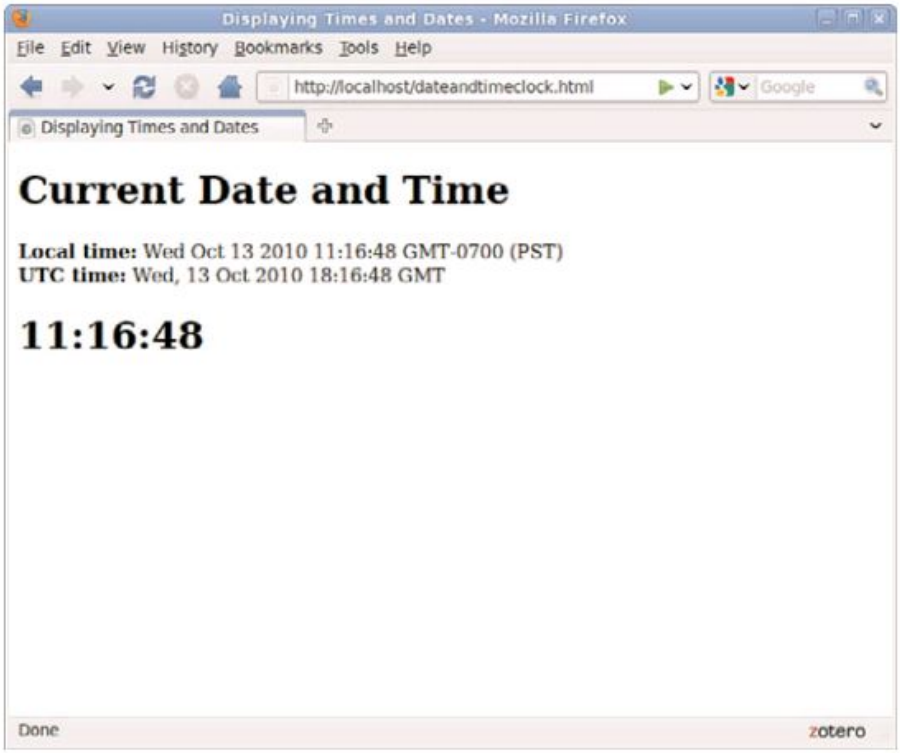


FIGURE 4.3
Firefox displays the modified Date and Time script.

NOTE

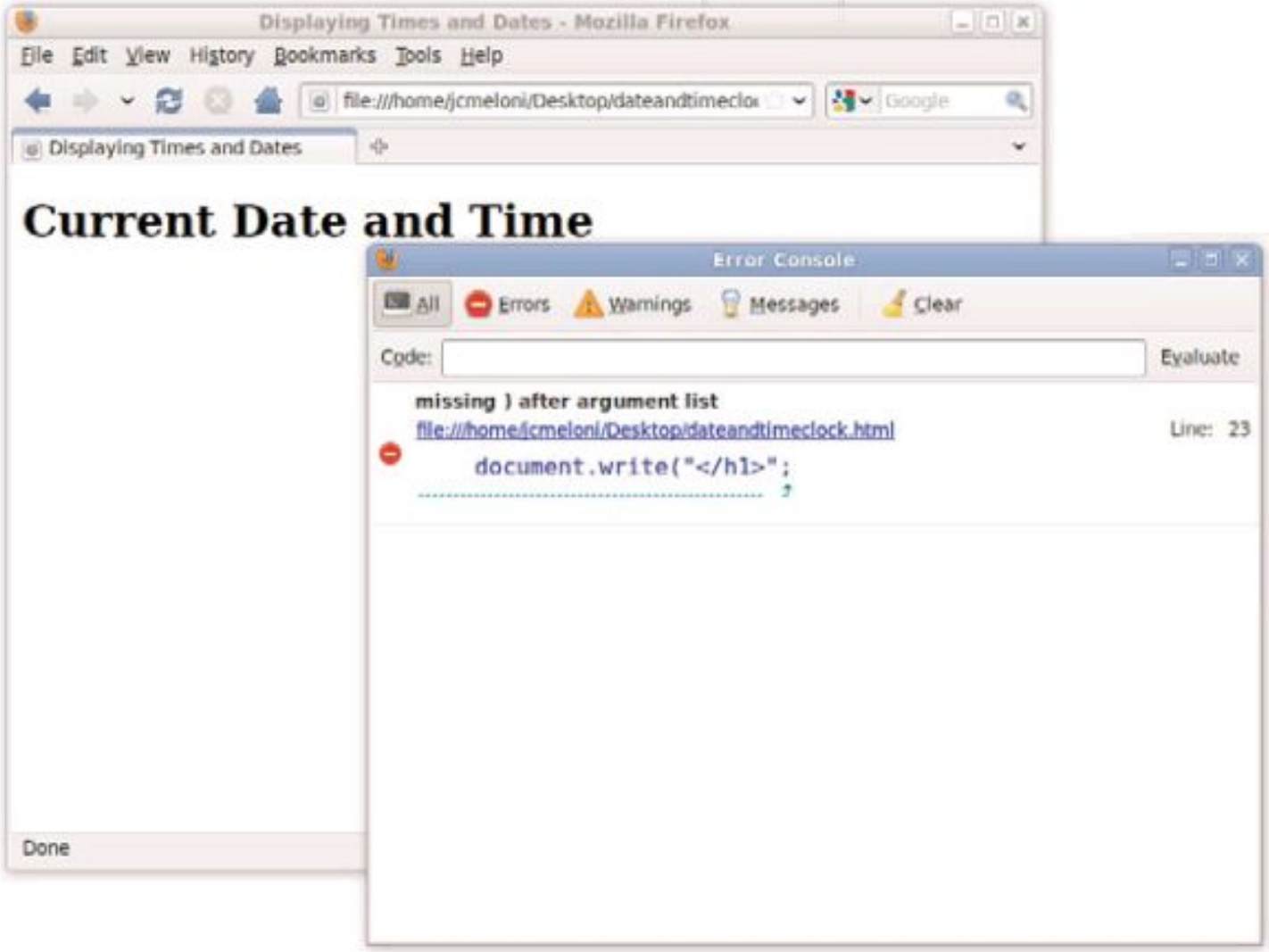
The time formatting produced by this script isn't perfect; hours after noon are in 24-hour time, and there are no leading zeroes, so 12:04 is displayed as 12:4. See Chapter 17 for solutions to these issues.

Save your HTML document again and load the document into the browser. Depending on the browser version you're using, one of two things will happen: Either an error message will be displayed, or the script will simply fail to execute.

If an error message is displayed, you're halfway to fixing the problem by adding the missing parenthesis. If no error was displayed, you should configure your browser to display error messages so that you can diagnose future problems:

- In Firefox, you can also select Tools, JavaScript Console from the menu. The console is shown in Figure 4.4, displaying the error message you created in this example.
- In Chrome, select Tools, JavaScript Console from the Customizations (Options) menu. A console will display in the bottom of the browser window.
- In Internet Explorer, select Tools, Internet Options. On the Advanced page, uncheck the Disable Script Debugging box and check the Display a Notification About Every Script Error box. (If this is disabled, a yellow icon in the status bar will still notify you of errors.)

The error we get in this case is `missing ) after argument list` (Firefox) or `Expected ')'` (Internet Explorer), which turns out to be exactly the problem. Be warned, however, that error messages aren't always this enlightening.



Although Internet Explorer displays error dialog boxes for each error, Firefox's JavaScript Console displays a single list of errors and enables you to test commands. For this reason, you might find it useful to install Firefox for debugging and testing JavaScript, even if Internet Explorer is your primary browser.