Chapter 1. Introduction to JavaScript

JavaScript is the programming language of the Web. The overwhelming majority of modern websites use JavaScript, and all modern web browsers—on desktops, game consoles, tablets, and smart phones—include JavaScript interpreters, making JavaScript the most ubiquitous programming language in history. JavaScript is part of the triad of technologies that all Web developers must learn: HTML to specify the content of web pages, CSS to specify the presentation of web pages, and JavaScript to specify the behavior of web pages. This book will help you master the language.

If you are already familiar with other programming languages, it may help you to know that JavaScript is a high-level, dynamic, untyped interpreted programming language that is well-suited to object-oriented and functional programming styles. JavaScript derives its syntax from Java, its first-class functions from Scheme, and its prototype-based inheritance from Self. But you do not need to know any of those languages, or be familiar with those terms, to use this book and learn JavaScript.

The name "JavaScript" is actually somewhat misleading. Except for a superficial syntactic resemblance, JavaScript is completely different from the Java programming language. And JavaScript has long since outgrown its scripting-language roots to become a robust and efficient general-purpose language. The latest version of the language (see the sidebar) defines new features for serious large-scale software development.

JAVASCRIPT: NAMES AND VERSIONS

JavaScript was created at Netscape in the early days of the Web, and technically, “JavaScript” is a trademark licensed from Sun Microsystems (now Oracle) used to describe Netscape’s (now Mozilla’s) implementation of the language. Netscape submitted the language for standardization to ECMA—the European Computer Manufacturer’s Association—and because of trademark issues, the standardized version of the language was stuck with the awkward name “ECMAScript.” For the same trademark reasons, Microsoft’s version of the language is formally known as “JScript.” In practice, just about everyone calls the language JavaScript. This book uses the name “ECMAScript” only to refer to the language standard.

For the last decade, all web browsers have implemented version 3 of the ECMAScript standard and there has really been no need to think about version numbers: the language standard was stable and browser implementations of the language were, for the most part, interoperable. Recently, an important new version of the language has been defined as ECMAScript version 5 and, at the time of this writing, browsers are beginning to implement it. This book covers all the new features of ECMAScript 5 as well as all the long-standing features of ECMAScript 3. You’ll sometimes see these language versions abbreviated as ES3 and ES5, just as you’ll sometimes see the name JavaScript abbreviated as JS.

When we’re speaking of the language itself, the only version numbers that are relevant are ECMAScript versions 3 or 5. (Version 4 of ECMAScript was under development for years, but proved to be too ambitious and was never released.) Sometimes, however, you’ll also see a JavaScript version number, such as JavaScript 1.5 or JavaScript 1.8. These are Mozilla’s version numbers: version 1.5 is basically ECMAScript 3, and later versions include nonstandard language extensions (see Chapter 11). Finally, there are also version numbers attached to particular JavaScript interpreters or “engines.” Google calls its JavaScript interpreter V8, for example, and at the time of this writing the current version of the V8 engine is 3.0.

To be useful, every language must have a platform or standard library or API of functions for performing things like basic input and output. The core JavaScript language defines a minimal API for working with text, arrays, dates, and regular expressions but does not include any input or output functionality. Input and output (as well as more sophisticated features, such as networking, storage, and graphics) are the responsibility of the "host environment" within which JavaScript is embedded. Usually that host environment is a web browser (though we'll see two uses of JavaScript without a web browser in Chapter 12). Part I of this book covers the language itself and its minimal built-in API. Part II explains how JavaScript is used in web browsers and covers the sprawling browser-based APIs loosely known as "client-side JavaScript."

Part III is the reference section for the core API. You can read about the JavaScript array manipulation API by looking up "Array" in this part of the book, for example. Part IV is the reference section for client-side JavaScript. You might look up "Canvas" in this part of the book to read about the graphics API defined by the HTML5 `<canvas>` element, for example.

This book covers low-level fundamentals first, and then builds on those to more advanced and higher-level abstractions. The chapters are intended to be read more or less in order. But learning a new programming language is never a linear process, and describing a language is not linear either: each language feature is related to other features and this book is full of cross-references—sometimes backward and sometimes forward to material you have not yet read. This chapter makes a quick first pass through the core language and the client-side API, introducing key features that will make it easier to understand the in-depth treatment in the chapters that follow.

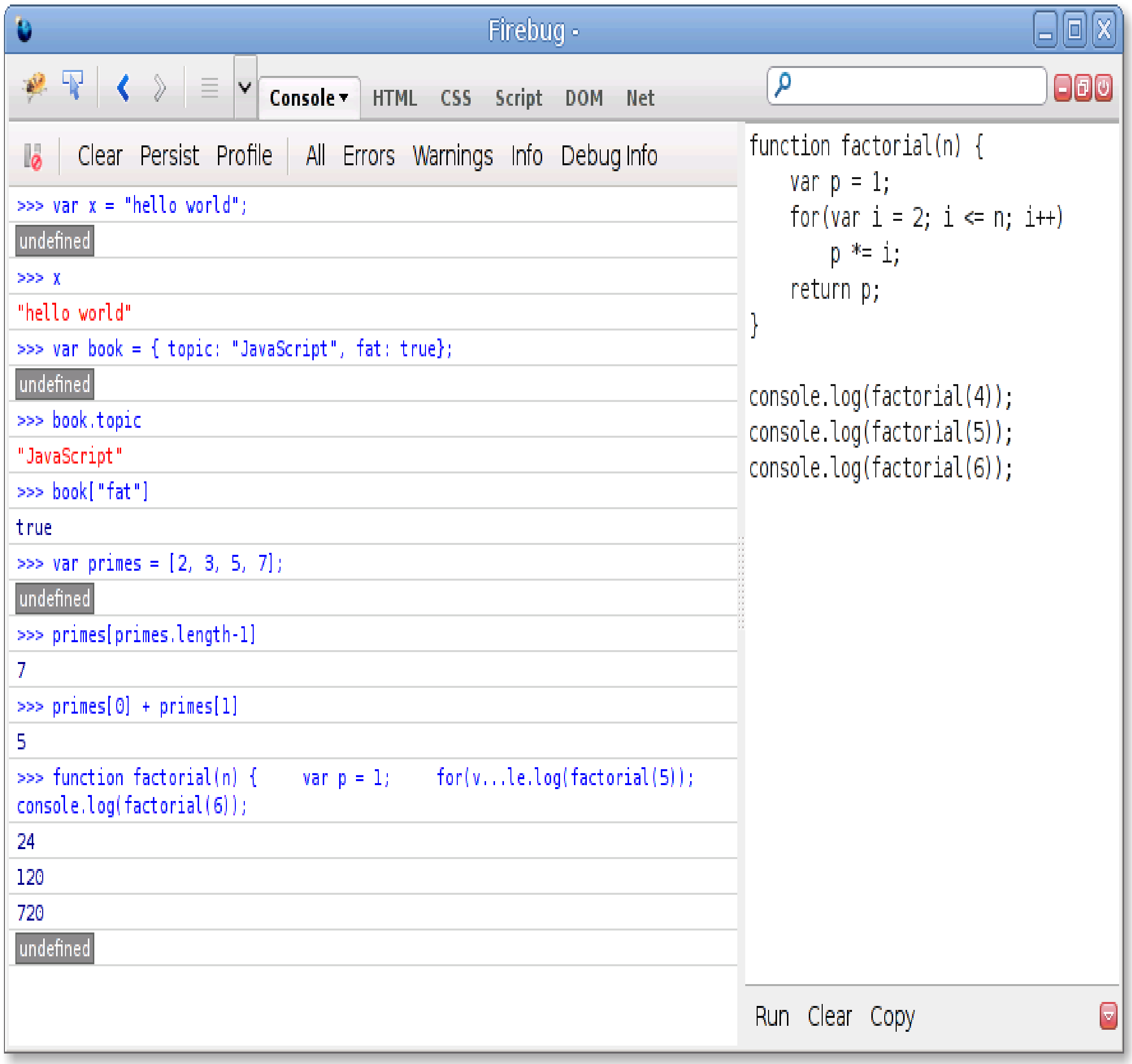
EXPLORING JAVASCRIPT

When learning a new programming language, it’s important to try the examples in the book, and then modify them and try them again to test your understanding of the language. To do that, you need a JavaScript interpreter. Fortunately, every web browser includes a JavaScript interpreter, and if you’re reading this book, you probably already have more than one web browser installed on your computer.

We’ll see later on in this chapter that you can embed JavaScript code within `<script>` tags in HTML files, and when the browser loads the file, it will execute the code. Fortunately, however, you don’t have to do that every time you want to try out simple snippets of JavaScript code. Spurred on by the powerful and innovative Firebug extension for Firefox (pictured in Figure 1-1 and available for download from http://getfirebug.com/), today’s web browsers all include web developer tools that are indispensable for debugging, experimenting, and learning. You can usually find these tools in the Tools menu of the browser under names like “Developer Tools” or “Web Console.” (Firefox 4 includes a built-in “Web Console,” but at the time of this writing, the Firebug extension is better.) Often, you can call up a console with a keystroke like F12 or Ctrl-Shift-J. These console tools often appear as panes at the top or bottom of the browser window, but some allow you to open them as separate windows (as pictured in Figure 1-1), which is often quite convenient.

A typical “developer tools” pane or window includes multiple tabs that allow you to inspect things like HTML document structure, CSS styles, network requests, and so on. One of the tabs is a “JavaScript console” that allows you to type in lines of JavaScript code and try them out. This is a particularly easy way to play around with JavaScript, and I recommend that you use it as you read this book.

There is a simple console API that is portably implemented by modern browsers. You can use the function`console.log()` to display text on the console. This is often surprisingly helpful while debugging, and some of the examples in this book (even in the core language section) use `console.log()` to perform simple output. A similar but more intrusive way to display output or debugging messages is by passing a string of text to the`alert()` function, which displays it in a modal dialog box.



```
>>> var x = "hello world";
undefined
>>> x
"hello world"
>>> var book = { topic: "JavaScript", fat: true};
undefined
>>> book.topic
"JavaScript"
>>> book["fat"]
true
>>> var primes = [2, 3, 5, 7];
undefined
>>> primes[primes.length-1]
7
>>> primes[0] + primes[1]
5
>>> function factorial(n) {     var p = 1;     for(v...le.log(factorial(5));
console.log(factorial(6));
24
120
720
undefined
```

```
function factorial(n) {
    var p = 1;
    for(var i = 2; i <= n; i++)
        p *= i;
    return p;
}

console.log(factorial(4));
console.log(factorial(5));
console.log(factorial(6));
```

Figure 1-1. The Firebug debugging console for Firefox

Core JavaScript

This section is a tour of the JavaScript language, and also a tour of Part I of this book. After this introductory chapter, we dive into JavaScript at the lowest level: Chapter 2, Lexical Structure, explains things like JavaScript comments, semicolons, and the Unicode character set. Chapter 3, Types, Values, and Variables, starts to get more interesting: it explains JavaScript variables and the values you can assign to those variables. Here's some sample code to illustrate the highlights of those two chapters:

```
// Anything following double slashes is an English-language comment.
// Read the comments carefully: they explain the JavaScript code.

// variable is a symbolic name for a value.
// Variables are declared with the var keyword:
var x;                              // Declare a variable named x.

// Values can be assigned to variables with an = sign
x = 0;                              // Now the variable x has the value 0
x                                   // => 0: A variable evaluates to its value.

// JavaScript supports several types of values
x = 1;                              // Numbers.
x = 0.01;                           // Just one Number type for integers and
reals.
x = "hello world";                 // Strings of text in quotation marks.
x = 'JavaScript';                  // Single quote marks also delimit strings.
x = true;                           // Boolean values.
x = false;                          // The other Boolean value.
x = null;                           // Null is a special value that means "no
value".
x = undefined;                     // Undefined is like null.
```

Two other very important *types* that JavaScript programs can manipulate are objects and arrays. These are the subject of Chapter 6, Objects, and Chapter 7, Arrays, but they are so important that you'll see them many times before you reach those chapters.

```
// JavaScript's most important data type is the object.
// An object is a collection of name/value pairs, or a string to value map.
var book = {                       // Objects are enclosed in curly braces.
     topic: "JavaScript",         // The property "topic" has value "JavaScript".
     fat: true                     // The property "fat" has value true.
};                              // The curly brace marks the end of the
object.

// Access the properties of an object with . or []:
book.topic                     // => "JavaScript"
book["fat"]                    // => true: another way to access property
values.
book.author = "Flanagan";   // Create new properties by assignment.
book.contents = {};          // {} is an empty object with no properties.

// JavaScript also supports arrays (numerically indexed lists) of values:
var primes = [2, 3, 5, 7]; // An array of 4 values, delimited with [ and
].
primes[0]                      // => 2: the first element (index 0) of the
array.
primes.length                  // => 4: how many elements in the array.
primes[primes.length-1]     // => 7: the last element of the array.
primes[4] = 9;                 // Add a new element by assignment.
primes[4] = 11;                // Or alter an existing element by assignment.
var empty = [];                // [] is an empty array with no elements.
empty.length                   // => 0

// Arrays and objects can hold other arrays and objects:
var points = [                 // An array with 2 elements.
     {x:0, y:0},               // Each element is an object.
     {x:1, y:1}
];
var data = {                   // An object with 2 properties
  trial1: [[1,2], [3,4]],   // The value of each property is an array.
  trial2: [[2,3], [4,5]]    // The elements of the arrays are arrays.
};
```

The syntax illustrated above for listing array elements within square braces or mapping object property names to property values inside curly braces is known as an *initializer expression*, and it is just one of the topics ofChapter 4, Expressions and Operators. An *expression* is a phrase of JavaScript that can be *evaluated* to produce a value. The use of `.` and `[]` to refer to the value of an object property or array element is an expression, for example. You may have noticed in the code above that when an expression stands alone on a line, the comment that follows it begins with an arrow (`=>`) and the value of the expression. This is a convention that you’ll see throughout this book.

One of the most common ways to form expressions in JavaScript is to use *operators* like these:

```
// Operators act on values (the operands) to produce a new value.
// Arithmetic operators are the most common:
3 + 2                           // => 5: addition
3 - 2                           // => 1: subtraction
3 * 2                           // => 6: multiplication
3 / 2                           // => 1.5: division
points[1].x - points[0].x   // => 1: more complicated operands work, too
"3" + "2"                      // => "32": + adds numbers, concatenates
strings

// JavaScript defines some shorthand arithmetic operators
var count = 0;                 // Define a variable
count++;                        // Increment the variable
count--;                        // Decrement the variable
count += 2;                     // Add 2: same as count = count + 2;
count *= 3;                     // Multiply by 3: same as count = count * 3;
count                           // => 6: variable names are expressions, too.

// Equality and relational operators test whether two values are equal,
// unequal, less than, greater than, and so on. They evaluate to true or
false.
var x = 2, y = 3;              // These = signs are assignment, not equality
tests
x == y                          // => false: equality
x != y                          // => true: inequality
x < y                           // => true: less-than
x <= y                          // => true: less-than or equal
x > y                           // => false: greater-than
x >= y                          // => false: greater-than or equal
"two" == "three"              // => false: the two strings are different
"two" > "three"               // => true: "tw" is alphabetically greater
than "th"
false == (x > y)               // => true: false is equal to false

// Logical operators combine or invert boolean values
(x == 2) && (y == 3)          // => true: both comparisons are true. && is
AND
(x > 3) || (y < 3)            // => false: neither comparison is true. || is
OR
!(x == y)                       // => true: ! inverts a boolean value
```

If the phrases of JavaScript are expressions, then the full sentences are *statements*, which are the topic ofChapter 5, Statements. In the code above, the lines that end with semicolons are statements. (In the code below, you'll see multiline statements that do not end with semicolons.) There is actually a lot of overlap between statements and expressions. Roughly, an expression is something that computes a value but doesn't *do* anything: it doesn't alter the program state in any way. Statements, on the other hand, don't have a value (or don't have a value that we care about), but they do alter the state. You’ve seen variable declarations and assignment statements above. The other broad category of statement is *control structures*, such as conditionals and loops. Examples are below, after we cover functions.

A *function* is a named and parameterized block of JavaScript code that you define once, and can then invoke over and over again. Functions aren’t covered formally until Chapter 8, Functions, but like objects and arrays, you’ll see them many times before you get to that chapter. Here are some simple examples:

```
// Functions are parameterized blocks of JavaScript code that we can
invoke.
function plus1(x) {          // Define a function named "plus1" with
parameter "x"
    return x+1;              // Return a value one larger than the value
passed in
}                            // Functions are enclosed in curly braces

plus1(y)                     // => 4: y is 3, so this invocation returns
3+1

var square = function(x) { // Functions are values and can be assigned to
vars
    return x*x;              // Compute the function's value
};                           // Semicolon marks the end of the
assignment.

square(plus1(y))             // => 16: invoke two functions in one
expression
```

When we combine functions with objects, we get *methods*:

```
// When functions are assigned to the properties of an object, we call
// them "methods".  All JavaScript objects have methods:
var a = [];                  // Create an empty array
a.push(1,2,3);               // The push() method adds elements to an array
a.reverse();                 // Another method: reverse the order of
elements

// We can define our own methods, too. The "this" keyword refers to the
object
// on which the method is defined: in this case, the points array from
above.
points.dist = function() { // Define a method to compute distance between
points
    var p1 = this[0];        // First element of array we're invoked on
    var p2 = this[1];       // Second element of the "this" object
    var a = p2.x-p1.x;      // Difference in X coordinates
    var b = p2.y-p1.y;      // Difference in Y coordinates
    return Math.sqrt(a*a + // The Pythagorean theorem
                   b*b); // Math.sqrt() computes the square root
};
points.dist()                 // => 1.414: distance between our 2 points
```

Now, as promised, here are some functions whose bodies demonstrate common JavaScript control structure statements:

```
// JavaScript statements include conditionals and loops using the syntax
// of C, C++, Java, and other languages.
function abs(x) {             // A function to compute the absolute value
    if (x >= 0) {             // The if statement...
        return x;             //   executes this code if the comparison is
true.
    }                          // This is the end of the if clause.
    else {                    // The optional else clause executes its code
if
        return -x;            //   the comparison is false.
    }                          // Curly braces optional when 1 statement
per clause.
}                              // Note return statements nested inside
if/else.

function factorial(n) {      // A function to compute factorials
    var product = 1;          // Start with a product of 1
    while(n > 1) {            // Repeat statements in {} while expr in () is
true
        product *= n;         // Shortcut for product = product * n;
        n--;                   // Shortcut for n = n - 1
    }                          // End of loop
    return product;           // Return the product
}
factorial(4)                  // => 24: 1*4*3*2

function factorial2(n) {     // Another version using a different loop
    var i, product = 1;      // Start with 1
    for(i=2; i <= n; i++)    // Automatically increment i from 2 up to n
        product *= i;         // Do this each time. {} not needed for 1-line
loops
    return product;           // Return the factorial
}
factorial2(5)                 // => 120: 1*2*3*4*5
```

JavaScript is an object-oriented language, but it is quite different than most. Chapter 9, Classes and Modules, covers object-oriented programming in JavaScript in detail, with lots of examples, and is one of the longest chapters in the book. Here is a very simple example that demonstrates how to define a JavaScript class to represent 2D geometric points. Objects that are instances of this class have a single method named `r()` that computes the distance of the point from the origin:

```
// Define a constructor function to initialize a new Point object
function Point(x,y) {       // By convention, constructors start with
capitals
    this.x = x;              // this keyword is the new object being
initialized
    this.y = y;              // Store function arguments as object
properties
}                             // No return is necessary

// Use a constructor function with the keyword "new" to create instances
var p = new Point(1, 1);    // The geometric point (1,1)

// Define methods for Point objects by assigning them to the prototype
// object associated with the constructor function.
Point.prototype.r = function() {
    return Math.sqrt(       // Return the square root of x² + y²
        this.x * this.x +   // This is the Point object on which the
method...
        this.y * this.y     // ...is invoked.
    );
};

// Now the Point object p (and all future Point objects) inherits the
method r()
p.r()                        // => 1.414...
```

Chapter 9 is really the climax of Part I, and the chapters that follow wrap up some loose ends and bring our exploration of the core language to a close. Chapter 10, Pattern Matching with Regular Expressions, explains the regular expression grammar and demonstrates how to use these “regexps” for textual pattern matching.Chapter 11, JavaScript Subsets and Extensions, covers subsets and extensions of core JavaScript. Finally, before we plunge into client-side JavaScript in web browsers, Chapter 12, Server-Side JavaScript, introduces two ways to use JavaScript outside of web browsers.

Client-Side JavaScript

Client-side JavaScript does not exhibit the nonlinear cross-reference problem nearly to the extent that the core language does, and it is possible to learn how to use JavaScript in web browsers in a fairly linear sequence. But you're probably reading this book to learn client-side JavaScript, and Part II is a long way off, so this section is a quick sketch of basic client-side programming techniques, followed by an in-depth example.

Chapter 13, JavaScript in Web Browsers, is the first chapter of Part II and it explains in detail how to put JavaScript to work in web browsers. The most important thing you'll learn in that chapter is that JavaScript code can be embedded within HTML files using the `<script>` tag:

```
<html>
<head>
<script src="library.js"></script> <!-- include a library of JavaScript
code -->
</head>
<body>
<p>This is a paragraph of HTML</p>
<script>
// And this is some client-side JavaScript code
// literally embedded within the HTML file
</script>
<p>Here is more HTML.</p>
</body>
</html>
```

Chapter 14, The Window Object, explains techniques for scripting the web browser and covers some important global functions of client-side JavaScript. For example:

```
<script>
function moveon() {
    // Display a modal dialog to ask the user a question
    var answer = confirm("Ready to move on?");
    // If they clicked the "OK" button, make the browser load a new page
    if (answer) window.location = "http://google.com";
}
```

```
// Run the function defined above 1 minute (60,000 milliseconds) from now.
setTimeout(moveon, 60000);
</script>
```

Note that the client-side example code shown in this section comes in longer snippets than the core language examples earlier in the chapter. These examples are not designed to be typed into a Firebug (or similar) console window. Instead you can embed them in an HTML file and try them out by loading them in your web browser. The code above, for instance, works as a stand-alone HTML file.

Chapter 15, Scripting Documents, gets down to the real business of client-side JavaScript, scripting HTML document content. It shows you how to select particular HTML elements from within a document, how to set HTML attributes of those elements, how to alter the content of those elements, and how to add new elements to the document. This function demonstrates a number of these basic document searching and modification techniques:

```
// Display a message in a special debugging output section of the document.
// If the document does not contain such a section, create one.
function debug(msg) {
     // Find the debugging section of the document, looking at HTML id
attributes
     var log = document.getElementById("debuglog");

     // If no element with the id "debuglog" exists, create one.
     if (!log) {
          log = document.createElement("div");   // Create a new <div>
element
          log.id = "debuglog";                     // Set the HTML id
attribute on it
          log.innerHTML = "<h1>Debug Log</h1>"; // Define initial content
          document.body.appendChild(log);        // Add it at end of
document
     }

     // Now wrap the message in its own <pre> and append it to the log
     var pre = document.createElement("pre");   // Create a <pre> tag
     var text = document.createTextNode(msg);   // Wrap msg in a text node
     pre.appendChild(text);                       // Add text to the <pre>
     log.appendChild(pre);                        // Add <pre> to the log
}
```

Chapter 15 shows how JavaScript can script the HTML elements that define web content. Chapter 16, Scripting CSS, shows how you can use JavaScript with the CSS styles that define the presentation of that content. This is often done with the `style` or `class` attribute of HTML elements:

```
function hide(e, reflow) { // Hide the element e by scripting its style
     if (reflow) {                              // If 2nd argument is true
          e.style.display = "none"          // hide element and use its space
     }
     else {                                      // Otherwise
          e.style.visibility = "hidden"; // make e invisible, but leave its
space
     }
}

function highlight(e) {      // Highlight e by setting a CSS class
     // Simply define or append to the HTML class attribute.
     // This assumes that a CSS stylesheet already defines the "hilite"
class
     if (!e.className) e.className = "hilite";
     else e.className += " hilite";
}
```

JavaScript allows us to script the HTML content and CSS presentation of documents in web browsers, but it also allows us to define behavior for those documents with *event handlers*. An event handler is a JavaScript function that we register with the browser and the browser invokes when some specified type of event occurs. The event of interest might be a mouse click or a key press (or on a smart phone, it might be a two-finger gesture of some sort). Or an event handler might be triggered when the browser finishes loading a document, when the user resizes the browser window, or when the user enters data into an HTML form element. Chapter 17, Handling Events, explains how you can define and register event handlers and how the browser invokes them when events occur.

The simplest way to define event handlers is with HTML attributes that begin with “on”. The “onclick” handler is a particularly useful one when you’re writing simple test programs. Suppose that you had typed in the `debug()` and `hide()` functions from above and saved them in files named *debug.js* and *hide.js*. You could write a simple HTML test file using `<button>` elements with `onclick` event handler attributes:

```
<script src="debug.js"></script>
<script src="hide.js"></script>
Hello
<button onclick="hide(this,true); debug('hide button 1');">Hide1</button>
<button onclick="hide(this); debug('hide button 2');">Hide2</button>
World
```

Here is some more client-side JavaScript code that uses events. It registers an event handler for the very important "load" event, and it also demonstrates a more sophisticated way of registering event handler functions for "click" events:

```
// The "load" event occurs when a document is fully loaded. Usually we
// need to wait for this event before we start running our JavaScript code.
window.onload = function() {   // Run this function when the document loads
     // Find all <img> tags in the document
     var images = document.getElementsByTagName("img");

     // Loop through them, adding an event handler for "click" events to
each
     // so that clicking on the image hides it.
     for(var i = 0; i < images.length; i++) {
          var image = images[i];
          if (image.addEventListener) // Another way to register a handler
               image.addEventListener("click", hide, false);
          else                                // For compatibility with IE8 and
before
               image.attachEvent("onclick", hide);
     }

     // This is the event handler function registered above
     function hide(event) { event.target.style.visibility = "hidden"; }
};
```

Chapters 15, 16, and 17 explain how you can use JavaScript to script the content (HTML), presentation (CSS), and behavior (event handling) of web pages. The APIs described in those chapters are somewhat complex and, until recently, riddled with browser incompatibilities. For these reasons, many or most client-side JavaScript programmers choose to use a client-side library or framework to simplify their basic programming tasks. The most popular such library is jQuery, the subject of Chapter 19, The jQuery Library . jQuery defines a clever and easy-to-use API for scripting document content, presentation, and behavior. It has been thoroughly tested and works in all major browsers, including old ones like IE6.

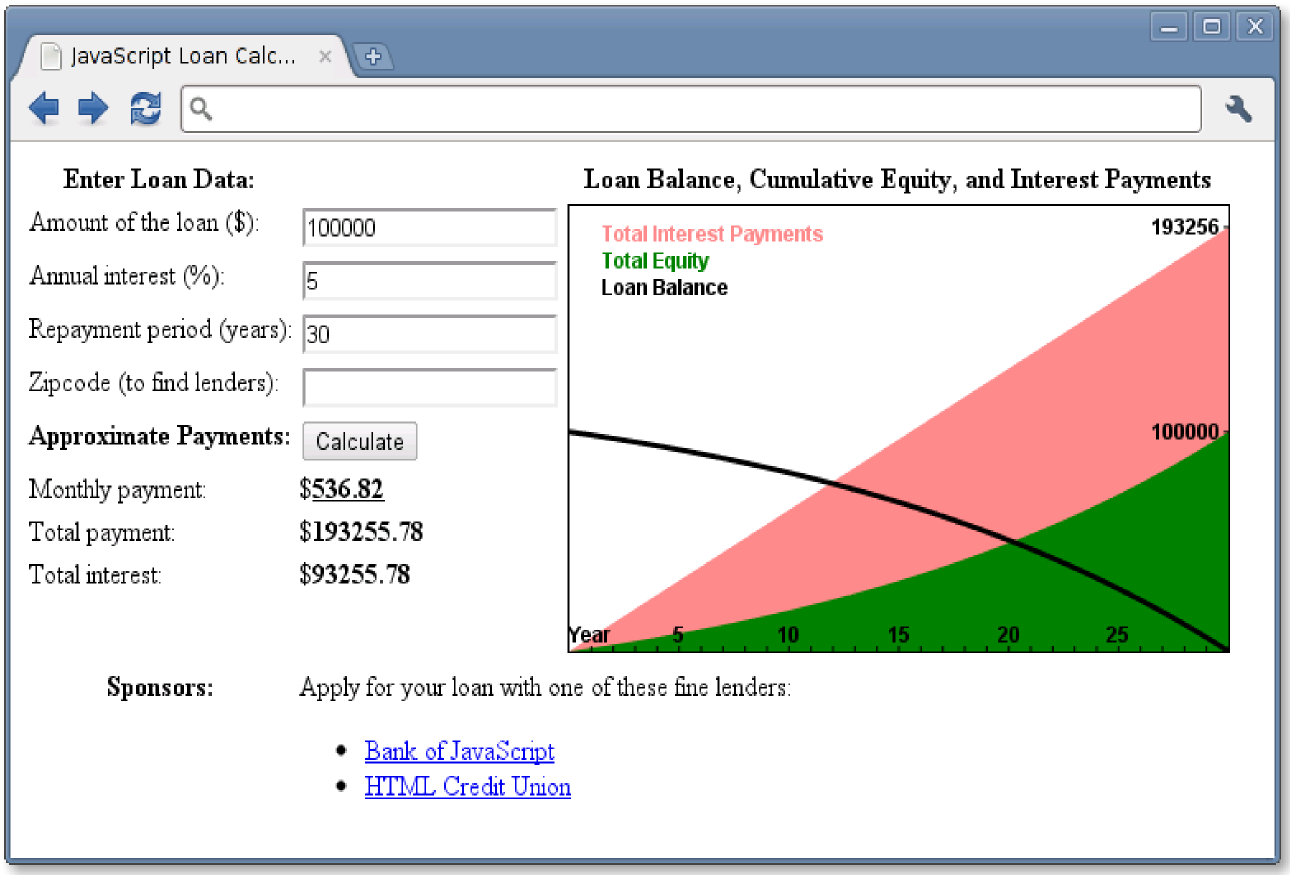
jQuery code is easy to identify because it makes frequent use of a function named `$()`. Here is what the`debug()` function used previously looks like when rewritten to use jQuery:

```
function debug(msg) {
    var log = $("#debuglog");           // Find the element to display
msg in.
    if (log.length == 0) {              // If it doesn't exist yet,
create it...
        log = $("<div id='debuglog'><h1>Debug Log</h1></div>");
        log.appendTo(document.body);    // and insert it at the end of the
body.
    }
    log.append($("<pre/>").text(msg)); // Wrap msg in <pre> and append to
log.
}
```

The four chapters of Part II described so far have all really been about web *pages*. Four more chapters shift gears to focus on web *applications*. These chapters are not about using web browsers to display documents with scriptable content, presentation, and behavior. Instead, they’re about using web browsers as application platforms, and they describe the APIs that modern browsers provide to support sophisticated client-side web apps. Chapter 18, Scripted HTTP, explains how to make scripted HTTP requests with JavaScript—a kind of networking API. Chapter 20, Client-Side Storage, describes mechanisms for storing data—and even entire applications—on the client side for use in future browsing sessions. Chapter 21, Scripted Media and Graphics, covers a client-side API for drawing arbitrary graphics in an HTML `<canvas>` tag. And, finally, Chapter 22, HTML5 APIs, covers an assortment of new web app APIs specified by or affiliated with HTML5. Networking, storage, graphics: these are OS-type services being provided by the web browser, defining a new cross-platform application environment. If you are targeting browsers that support these new APIs, it is an exciting time to be a client-side JavaScript programmer. There are no code samples from these final four chapters here, but the extended example below uses some of these new APIs.

Example: A JavaScript Loan Calculator

This chapter ends with an extended example that puts many of these techniques together and shows what real-world client-side JavaScript (plus HTML and CSS) programs look like. Example 1-1 lists the code for the simple loan payment calculator application pictured in Figure 1-2.



Enter Loan Data:

Amount of the loan ($):	100000
Annual interest (%):	5
Repayment period (years):	30
Zipcode (to find lenders):	
Approximate Payments:	
Monthly payment:	**$536.82**
Total payment:	**$193255.78**
Total interest:	**$93255.78**

Sponsors: Apply for your loan with one of these fine lenders:

- Bank of JavaScript
- HTML Credit Union

Figure 1-2. A loan calculator web application

It is worth reading through Example 1-1 carefully. You shouldn’t expect to understand everything, but the code is heavily commented and you should be able to at least get the big-picture view of how it works. The example demonstrates a number of core JavaScript language features, and also demonstrates important client-side JavaScript techniques:

- How to find elements in a document.
- How to get user input from form input elements.
- How to set the HTML content of document elements.
- How to store data in the browser.
- How to make scripted HTTP requests.
- How to draw graphics with the `<canvas>` element.

Example 1-1. A loan calculator in JavaScript

```
<!DOCTYPE html>
<html>
<head>
<title>JavaScript Loan Calculator</title>
<style> /* This is a CSS style sheet: it adds style to the program output
*/
.output { font-weight: bold; }              /* Calculated values in bold */
#payment { text-decoration: underline; } /* For element with id="payment"
*/
#graph { border: solid black 1px; }        /* Chart has a simple border */
th, td { vertical-align: top; }             /* Don't center table cells */
</style>
```

```
</head>

<body>

<!--

  This is an HTML table with <input> elements that allow the user to enter
data

  and <span> elements in which the program can display its results.

  These elements have ids like "interest" and "years". These ids are used

  in the JavaScript code that follows the table. Note that some of the
input

  elements define "onchange" or "onclick" event handlers. These specify
strings

  of JavaScript code to be executed when the user enters data or clicks.

-->

<table>

  <tr><th>Enter Loan Data:</th>

      <td></td>

      <th>Loan Balance, Cumulative Equity, and Interest
Payments</th></tr>

  <tr><td>Amount of the loan ($):</td>

      <td><input id="amount" onchange="calculate();"></td>

      <td rowspan=8>

         <canvas id="graph" width="400" height="250"></canvas></td></tr>

  <tr><td>Annual interest (%):</td>

      <td><input id="apr" onchange="calculate();"></td></tr>

  <tr><td>Repayment period (years):</td>

      <td><input id="years" onchange="calculate();"></td>

  <tr><td>Zipcode (to find lenders):</td>
```

```
        <td><input id="zipcode" onchange="calculate();"></td>
    <tr><th>Approximate Payments:</th>
        <td><button onclick="calculate();">Calculate</button></td></tr>
    <tr><td>Monthly payment:</td>
        <td>$<span class="output" id="payment"></span></td></tr>
    <tr><td>Total payment:</td>
        <td>$<span class="output" id="total"></span></td></tr>
    <tr><td>Total interest:</td>
        <td>$<span class="output" id="totalinterest"></span></td></tr>
    <tr><th>Sponsors:</th><td  colspan=2>
      Apply for your loan with one of these fine lenders:
      <div id="lenders"></div></td></tr>
</table>

<!-- The rest of this example is JavaScript code in the <script> tag below
-->
<!-- Normally, this script would go in the document <head> above but it -->
<!-- is easier to understand here, after you've seen its HTML context. -->
<script>
"use strict"; // Use ECMAScript 5 strict mode in browsers that support it

/*
 * This script defines the calculate() function called by the event
handlers
 * in HTML above. The function reads values from <input> elements,
calculates
```

```
 * loan payment information, displays the results in <span> elements. It
also
 * saves the user's data, displays links to lenders, and draws a chart.
 */
function calculate() {
    // Look up the input and output elements in the document
    var amount = document.getElementById("amount");
    var apr = document.getElementById("apr");
    var years = document.getElementById("years");
    var zipcode = document.getElementById("zipcode");
    var payment = document.getElementById("payment");
    var total = document.getElementById("total");
    var totalinterest = document.getElementById("totalinterest");

    // Get the user's input from the input elements. Assume it is all
valid.
    // Convert interest from a percentage to a decimal, and convert from
    // an annual rate to a monthly rate. Convert payment period in years
    // to the number of monthly payments.
    var principal = parseFloat(amount.value);
    var interest = parseFloat(apr.value) / 100 / 12;
    var payments = parseFloat(years.value) * 12;

    // Now compute the monthly payment figure.
    var x = Math.pow(1 + interest, payments);   // Math.pow() computes
powers
    var monthly = (principal*x*interest)/(x-1);
    // If the result is a finite number, the user's input was good and
    // we have meaningful results to display
    if (isFinite(monthly)) {
        // Fill in the output fields, rounding to 2 decimal places
        payment.innerHTML = monthly.toFixed(2);
        total.innerHTML = (monthly * payments).toFixed(2);
        totalinterest.innerHTML = ((monthly*payments)-
principal).toFixed(2);

        // Save the user's input so we can restore it the next time they
visit
        save(amount.value, apr.value, years.value, zipcode.value);

        // Advertise: find and display local lenders, but ignore network
errors
        try {      // Catch any errors that occur within these curly
braces
            getLenders(amount.value, apr.value, years.value,
zipcode.value);
        }
        catch(e) { /* And ignore those errors */ }

        // Finally, chart loan balance, and interest and equity payments
        chart(principal, interest, monthly, payments);
    }
    else {
          // Result was Not-a-Number or infinite, which means the input was
          // incomplete or invalid. Clear any previously displayed output.
          payment.innerHTML = "";          // Erase the content of these
elements
          total.innerHTML = ""
          totalinterest.innerHTML = "";
          chart();                          // With no arguments, clears
the chart
     }
}

// Save the user's input as properties of the localStorage object. Those
// properties will still be there when the user visits in the future
// This storage feature will not work in some browsers (Firefox, e.g.) if
you
// run the example from a local file:// URL.  It does work over HTTP,
however.
function save(amount, apr, years, zipcode) {
     if (window.localStorage) {   // Only do this if the browser supports it
          localStorage.loan_amount = amount;
          localStorage.loan_apr = apr;
          localStorage.loan_years = years;
          localStorage.loan_zipcode = zipcode;
     }
}
```

```
// Automatically attempt to restore input fields when the document first
loads.
window.onload = function() {
    // If the browser supports localStorage and we have some stored data
    if (window.localStorage && localStorage.loan_amount) {
        document.getElementById("amount").value =
localStorage.loan_amount;
        document.getElementById("apr").value = localStorage.loan_apr;
        document.getElementById("years").value = localStorage.loan_years;
        document.getElementById("zipcode").value =
localStorage.loan_zipcode;
    }
};

// Pass the user's input to a server-side script which can (in theory)
return
// a list of links to local lenders interested in making loans.  This
example
// does not actually include a working implementation of such a lender-
finding
// service. But if the service existed, this function would work with it.
function getLenders(amount, apr, years, zipcode) {
    // If the browser does not support the XMLHttpRequest object, do
nothing
    if (!window.XMLHttpRequest) return;

    // Find the element to display the list of lenders in
    var ad = document.getElementById("lenders");
    if (!ad) return;                                 // Quit if no spot for
output

    // Encode the user's input as query parameters in a URL
    var url = "getLenders.php" +                    // Service url plus
        "?amt=" + encodeURIComponent(amount) +   // user data in query
string
        "&apr=" + encodeURIComponent(apr) +
        "&yrs=" + encodeURIComponent(years) +
        "&zip=" + encodeURIComponent(zipcode);

    // Fetch the contents of that URL using the XMLHttpRequest object
    var req = new XMLHttpRequest();          // Begin a new request
    req.open("GET", url);                      // An HTTP GET request for
the url
    req.send(null);                             // Send the request with no
body

    // Before returning, register an event handler function that will be
called
    // at some later time when the HTTP server's response arrives. This
kind of
    // asynchronous programming is very common in client-side JavaScript.
    req.onreadystatechange = function() {
        if (req.readyState == 4 && req.status == 200) {
            // If we get here, we got a complete valid HTTP response
            var response = req.responseText;     // HTTP response as a
string
            var lenders = JSON.parse(response);  // Parse it to a JS
array

            // Convert the array of lender objects to a string of HTML
            var list = "";
            for(var i = 0; i < lenders.length; i++) {
                list += "<li><a href='" + lenders[i].url + "'>" +
                    lenders[i].name + "</a>";
            }

            // Display the HTML in the element from above.
            ad.innerHTML = "<ul>" + list + "</ul>";
        }
    }
}

// Chart monthly loan balance, interest and equity in an HTML <canvas>
element.
// If called with no arguments then just erase any previously drawn chart.
function chart(principal, interest, monthly, payments) {
    var graph = document.getElementById("graph"); // Get the <canvas> tag
    graph.width = graph.width;  // Magic to clear and reset the canvas
element
```

```
    // If we're called with no arguments, or if this browser does not
support
    // graphics in a <canvas> element, then just return now.
    if (arguments.length == 0 || !graph.getContext) return;

    // Get the "context" object for the <canvas> that defines the drawing
API
    var g = graph.getContext("2d"); // All drawing is done with this
object
    var width = graph.width, height = graph.height; // Get canvas size

    // These functions convert payment numbers and dollar amounts to
pixels
    function paymentToX(n) { return n * width/payments; }
    function amountToY(a) { return height-(a *
height/(monthly*payments*1.05));}

    // Payments are a straight line from (0,0) to (payments,
monthly*payments)
    g.moveTo(paymentToX(0), amountToY(0));          // Start at lower left
    g.lineTo(paymentToX(payments),                   // Draw to upper
right
             amountToY(monthly*payments));
    g.lineTo(paymentToX(payments), amountToY(0));  // Down to lower right
    g.closePath();                                    // And back to
start
    g.fillStyle = "#f88";                            // Light red
    g.fill();                                              // Fill the
triangle
    g.font = "bold 12px sans-serif";                   // Define a font
    g.fillText("Total Interest Payments", 20,20);    // Draw text in legend

    // Cumulative equity is non-linear and trickier to chart
    var equity = 0;
    g.beginPath();                                        // Begin a new
shape
    g.moveTo(paymentToX(0), amountToY(0));            // starting at lower-
left
    for(var p = 1; p <= payments; p++) {
        // For each payment, figure out how much is interest
        var thisMonthsInterest = (principal-equity)*interest;
        equity += (monthly - thisMonthsInterest);     // The rest goes to
equity
        g.lineTo(paymentToX(p),amountToY(equity)); // Line to this point
    }
    g.lineTo(paymentToX(payments), amountToY(0));    // Line back to X axis
    g.closePath();                                        // And back to
start point
    g.fillStyle = "green";                               // Now use green
paint
    g.fill();                                              // And fill area
under curve
    g.fillText("Total Equity", 20,35);                 // Label it in green
    // Loop again, as above, but chart loan balance as a thick black line
    var bal = principal;
    g.beginPath();
    g.moveTo(paymentToX(0),amountToY(bal));
    for(var p = 1; p <= payments; p++) {
        var thisMonthsInterest = bal*interest;
        bal -= (monthly - thisMonthsInterest);      // The rest goes to
equity
        g.lineTo(paymentToX(p),amountToY(bal));     // Draw line to this
point
    }
    g.lineWidth = 3;                                         // Use a thick
line
    g.stroke();                                              // Draw the
balance curve
    g.fillStyle = "black";                                   // Switch to black
text
    g.fillText("Loan Balance", 20,50);                  // Legend entry

    // Now make yearly tick marks and year numbers on X axis
    g.textAlign="center";                                    // Center text over
ticks
    var y = amountToY(0);                                    // Y coordinate of
X axis
    for(var year=1; year*12 <= payments; year++) {  // For each year
        var x = paymentToX(year*12);                      // Compute tick
position
```

```
          g.fillRect(x-0.5,y-3,1,3);                    // Draw the tick
          if (year == 1) g.fillText("Year", x, y-5); // Label the axis
          if (year % 5 == 0 && year*12 !== payments) // Number every 5
years
               g.fillText(String(year), x, y-5);
     }
     // Mark payment amounts along the right edge
     g.textAlign = "right";                              // Right-justify
text
     g.textBaseline = "middle";                          // Center it
vertically
     var ticks = [monthly*payments, principal];      // The two points
we'll mark
     var rightEdge = paymentToX(payments);           // X coordinate of Y
axis
     for(var i = 0; i < ticks.length; i++) {          // For each of the 2
points
          var y = amountToY(ticks[i]);                  // Compute Y
position of tick
          g.fillRect(rightEdge-3, y-0.5, 3,1);        // Draw the tick mark
          g.fillText(String(ticks[i].toFixed(0)),     // And label it.
                          rightEdge-5, y);
     }
}
</script>
</body>
</html>
```

Part I. Core JavaScript

This part of the book, Chapters 2 though 12, documents the core JavaScript language and is meant to be a JavaScript language reference. After you read through it once to learn the language, you may find yourself referring back to it to refresh your memory about some of the trickier points of JavaScript.

Chapter 2. Lexical Structure

The lexical structure of a programming language is the set of elementary rules that specifies how you write programs in that language. It is the lowest-level syntax of a language; it specifies such things as what variable names look like, the delimiter characters for comments, and how one program statement is separated from the next. This short chapter documents the lexical structure of JavaScript.

Character Set

JavaScript programs are written using the Unicode character set. Unicode is a superset of ASCII and Latin-1 and supports virtually every written language currently used on the planet. ECMAScript 3 requires JavaScript implementations to support Unicode version 2.1 or later, and ECMAScript 5 requires implementations to support Unicode 3 or later. See the sidebar in Text for more about Unicode and JavaScript.

Case Sensitivity

JavaScript is a case-sensitive language. This means that language keywords, variables, function names, and other*identifiers* must always be typed with a consistent capitalization of letters. The `while` keyword, for example, must be typed "while," not "While" or "WHILE." Similarly, `online`, `Online`, `OnLine`, and `ONLINE` are four distinct variable names.

Note, however, that HTML is not case-sensitive (although XHTML is). Because of its close association with client-side JavaScript, this difference can be confusing. Many client-side JavaScript objects and properties have the same names as the HTML tags and attributes they represent. While these tags and attribute names can be typed in any case in HTML, in JavaScript they typically must be all lowercase.

For example, the HTML`onclick` event handler attribute is sometimes specified as `onClick` in HTML, but it must be specified as`onclick` in JavaScript code (or in XHTML documents).

Whitespace, Line Breaks, and Format Control Characters

JavaScript ignores spaces that appear between tokens in programs. For the most part, JavaScript also ignores line breaks (but see Optional Semicolons for an exception). Because you can use spaces and newlines freely in your programs, you can format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

In addition to the regular space character (`\u0020`), JavaScript also recognizes the following characters as whitespace: tab (`\u0009`), vertical tab (`\u000B`), form feed (`\u000C`), nonbreaking space (`\u00A0`), byte order mark (`\uFEFF`), and any character in Unicode category Zs. JavaScript recognizes the following characters as line terminators: line feed (`\u000A`), carriage return (`\u000D`), line separator (`\u2028`), and paragraph separator (`\u2029`). A carriage return, line feed sequence is treated as a single line terminator.

Unicode format control characters (category Cf), such as RIGHT-TO-LEFT MARK (`\u200F`) and LEFT-TO-RIGHT MARK (`\u200E`), control the visual presentation of the text they occur in. They are important for the proper display of some non-English languages and are allowed in JavaScript comments, string literals, and regular expression literals, but not in the identifiers (e.g., variable names) of a JavaScript program. As a special case, ZERO WIDTH JOINER (`\u200D`) and ZERO WIDTH NON-JOINER (`\u200C`) are allowed in identifiers, but not as the first character. As noted above, the byte order mark format control character (`\uFEFF`) is treated as a space character.

Unicode Escape Sequences

Some computer hardware and software can not display or input the full set of Unicode characters. To support programmers using this older technology, JavaScript defines special sequences of six ASCII characters to represent any 16-bit Unicode codepoint. These Unicode escapes begin with the characters `\u` and are followed by exactly four hexadecimal digits (using uppercase or lowercase letters A–F). Unicode escapes may appear in JavaScript string literals, regular expression literals, and in identifiers (but not in language keywords). The Unicode escape for the character é, for example, is `\u00E9`, and the following two JavaScript strings are identical:

```
"café" === "caf\u00e9"    // => true
```

Unicode escapes may also appear in comments, but since comments are ignored, they are treated as ASCII characters in that context and not interpreted as Unicode.

Normalization

Unicode allows more than one way of encoding the same character. The string "é", for example, can be encoded as the single Unicode character `\u00E9` or as a regular ASCII e followed by the acute accent combining mark`\u0301`. These two encodings may look exactly the same when displayed by a text editor, but they have different binary encodings and are considered different by the computer. The Unicode standard defines the preferred encoding for all characters and specifies a normalization procedure to convert text to a canonical form suitable for comparisons. JavaScript assumes that the source code it is interpreting has already been normalized and makes no attempt to normalize identifiers, strings, or regular expressions itself.

Comments

JavaScript supports two styles of comments. Any text between a `//` and the end of a line is treated as a comment and is ignored by JavaScript. Any text between the characters `/*` and `*/` is also treated as a comment; these comments may span multiple lines but may not be nested. The following lines of code are all legal JavaScript comments:

```
// This is a single-line comment.
/* This is also a comment */  // and here is another comment.
/*
 * This is yet another comment.
 * It has multiple lines.
 */
```

Literals

A *literal* is a data value that appears directly in a program. The following are all literals:

```
12                   // The number twelve
1.2                  // The number one point two
"hello world"       // A string of text
'Hi'                 // Another string
true                 // A Boolean value
false                // The other Boolean value
/javascript/gi      // A "regular expression" literal (for pattern matching)
null                 // Absence of an object
```

Complete details on numeric and string literals appear in Chapter 3. Regular expression literals are covered inChapter 10. More complex expressions (see Object and Array Initializers) can serve as array and object literals. For example:

```
{ x:1, y:2 }      // An object initializer
[1,2,3,4,5]       // An array initializer
```

Identifiers and Reserved Words

An *identifier* is simply a name. In JavaScript, identifiers are used to name variables and functions and to provide labels for certain loops in JavaScript code. A JavaScript identifier must begin with a letter, an underscore (_), or a dollar sign (`$`). Subsequent characters can be letters, digits, underscores, or dollar signs. (Digits are not allowed as the first character so that JavaScript can easily distinguish identifiers from numbers.) These are all legal identifiers:

```
i
my_variable_name
v13
_dummy
$str
```

For portability and ease of editing, it is common to use only ASCII letters and digits in identifiers. Note, however, that JavaScript allows identifiers to contain letters and digits from the entire Unicode character set. (Technically, the ECMAScript standard also allows Unicode characters from the obscure categories Mn, Mc, and Pc to appear in identifiers after the first character.) This allows programmers to use variable names from non-English languages and also to use mathematical symbols:

```
var sí = true;
var π = 3.14;
```

Like any language, JavaScript reserves certain identifiers for use by the language itself. These “reserved words” cannot be used as regular identifiers. They are listed below.

Reserved Words

JavaScript reserves a number of identifiers as the keywords of the language itself. You cannot use these words as identifiers in your programs:

```
  break          delete         function       return
  typeof

  case           do             if             switch         var

  catch          else           in             this           void

  continue       false          instanceof     throw          while

  debugger       finally        new            true           with

  default        for            null           try
```

JavaScript also reserves certain keywords that are not currently used by the language but which might be used in future versions. ECMAScript 5 reserves the following words:

```
  class     const     enum      export    extends   import    super
```

In addition, the following words, which are legal in ordinary JavaScript code, are reserved in strict mode:

```
  implements     let            private        public         yield

  interface      package        protected      static
```

Strict mode also imposes restrictions on the use of the following identifiers. They are not fully reserved, but they are not allowed as variable, function, or parameter names:

```
arguments      eval
```

ECMAScript 3 reserved all the keywords of the Java language, and although this has been relaxed in ECMAScript 5, you should still avoid all of these identifiers if you plan to run your code under an ECMAScript 3 implementation of JavaScript:

```
abstract       double         goto           native
static

boolean        enum           implements     package        super

byte           export         import         private
synchronized

char           extends        int            protected
throws

class          final          interface      public
transient

const          float          long           short
volatile
```

JavaScript predefines a number of global variables and functions, and you should avoid using their names for your own variables and functions:

```
arguments           encodeURI           Infinity  Number
RegExp

Array               encodeURIComponent  isFinite  Object
String

Boolean             Error               isNaN     parseFloat
SyntaxError
```

```
Date                 eval                 JSON      parseInt
TypeError

decodeURI            EvalError            Math      RangeError
undefined

decodeURIComponent   Function             NaN       ReferenceError
URIError
```

Keep in mind that JavaScript implementations may define other global variables and functions, and each specific JavaScript embedding (client-side, server-side, etc.) will have its own list of global properties. See the Window object in Part IV for a list of the global variables and functions defined by client-side JavaScript.

Optional Semicolons

Like many programming languages, JavaScript uses the semicolon (`;`) to separate statements (see Chapter 5) from each other. This is important to make the meaning of your code clear: without a separator, the end of one statement might appear to be the beginning of the next, or vice versa. In JavaScript, you can usually omit the semicolon between two statements if those statements are written on separate lines. (You can also omit a semicolon at the end of a program or if the next token in the program is a closing curly brace `}`.) Many JavaScript programmers (and the code in this book) use semicolons to explicitly mark the ends of statements, even where they are not required. Another style is to omit semicolons whenever possible, using them only in the few situations that require them. Whichever style you choose, there are a few details you should understand about optional semicolons in JavaScript.

Consider the following code. Since the two statements appear on separate lines, the first semicolon could be omitted:

```
a = 3;
b = 4;
```

Written as follows, however, the first semicolon is required:

```
a = 3; b = 4;
```

Note that JavaScript does not treat every line break as a semicolon: it usually treats line breaks as semicolons only if it can't parse the code without the semicolons. More formally (and with two exceptions described below), JavaScript treats a line break as a semicolon if the next nonspace character cannot be interpreted as a continuation of the current statement. Consider the following code:

```
var a
a
=
3
console.log(a)
```

JavaScript interprets this code like this:

```
var a; a = 3; console.log(a);
```

JavaScript does treat the first line break as a semicolon because it cannot parse the code `var a a` without a semicolon. The second `a` could stand alone as the statement `a;`, but JavaScript does not treat the second line break as a semicolon because it can continue parsing the longer statement `a = 3;`.

These statement termination rules lead to some surprising cases. This code looks like two separate statements separated with a newline:

```
var y = x + f
(a+b).toString()
```

But the parentheses on the second line of code can be interpreted as a function invocation of `f` from the first line, and JavaScript interprets the code like this:

```
var y = x + f(a+b).toString();
```

More likely than not, this is not the interpretation intended by the author of the code. In order to work as two separate statements, an explicit semicolon is required in this case.

In general, if a statement begins with `(`, `[`, `/`, `+`, or `-`, there is a chance that it could be interpreted as a continuation of the statement before. Statements beginning with `/`, `+`, and `-` are quite rare in practice, but statements beginning with `(` and `[` are not uncommon at all, at least in some styles of JavaScript programming. Some programmers like to put a defensive semicolon at the beginning of any such statement so that it will continue to work correctly even if the statement before it is modified and a previously terminating semicolon removed:

```
var x = 0                           // Semicolon omitted here
;[x,x+1,x+2].forEach(console.log)  // Defensive ; keeps this statement
separate
```

There are two exceptions to the general rule that JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line. The first exception involves the `return`,`break`, and `continue` statements (see Chapter 5). These statements often stand alone, but they are sometimes followed by an identifier or expression. If a line break appears after any of these words (before any other tokens), JavaScript will always interpret that line break as a semicolon. For example, if you write:

```
return
true;
```

JavaScript assumes you meant:

```
return; true;
```

However, you probably meant:

```
return true;
```

What this means is that you must not insert a line break between `return`, `break` or `continue` and the expression that follows the keyword. If you do insert a line break, your code is likely to fail in a nonobvious way that is difficult to debug.

The second exception involves the `++` and `--` operators (Arithmetic Expressions). These operators can be prefix operators that appear before an expression or postfix operators that appear after an expression. If you want to use either of these operators as postfix operators, they must appear on the same line as the expression they apply to. Otherwise, the line break will be treated as a semicolon, and the `++` or `--` will be parsed as a prefix operator applied to the code that follows. Consider this code, for example:

```
x
++
y
```

It is parsed as `x; ++y;`, not as `x++; y`.

Chapter 3. Types, Values, and Variables

Computer programs work by manipulating *values*, such as the number 3.14 or the text “Hello World.” The kinds of values that can be represented and manipulated in a programming language are known as *types*, and one of the most fundamental characteristics of a programming language is the set of types it supports. When a program needs to retain a value for future use, it assigns the value to (or “stores” the value in) a *variable*. A variable defines a symbolic name for a value and allows the value to be referred to by name. The way that variables work is another fundamental characteristic of any programming language. This chapter explains types, values, and variables in JavaScript. These introductory paragraphs provide an overview, and you may find it helpful to refer to Core JavaScript while you read them. The sections that follow cover these topics in depth.

JavaScript types can be divided into two categories: *primitive types* and *object types*. JavaScript’s primitive types include numbers, strings of text (known as *strings*), and Boolean truth values (known as *booleans*). A significant portion of this chapter is dedicated to a detailed explanation of the numeric (Numbers) and string (Text) types in JavaScript. Booleans are covered in Boolean Values.

The special JavaScript values `null` and `undefined` are primitive values, but they are not numbers, strings, or booleans. Each value is typically considered to be the sole member of its own special type. null and undefinedhas more about `null` and `undefined`.

Any JavaScript value that is not a number, a string, a boolean, or `null` or `undefined` is an object. An object (that is, a member of the type *object*) is a collection of *properties* where each property has a name and a value (either a primitive value, such as a number or string, or an object). One very special object, the *global object*, is covered in The Global Object, but more general and more detailed coverage of objects is in Chapter 6.

An ordinary JavaScript object is an unordered collection of named values. The language also defines a special kind of object, known as an *array*, that represents an ordered collection of numbered values. The JavaScript language includes special syntax for working with arrays, and arrays have some special behavior that distinguishes them from ordinary objects. Arrays are the subject of Chapter 7.

JavaScript defines another special kind of object, known as a *function*. A function is an object that has executable code associated with it. A function may be *invoked* to run that executable code and return a computed value. Like arrays, functions behave differently from other kinds of objects, and JavaScript defines a special language syntax for working with them. The most important thing about functions in JavaScript is that they are true values and that JavaScript programs can treat them like regular objects. Functions are covered in Chapter 8.

Functions that are written to be used (with the `new` operator) to initialize a newly created object are known as*constructors*. Each constructor defines a *class* of objects—the set of objects initialized by that constructor. Classes can be thought of as subtypes of the object type. In addition to the Array and Function classes, core JavaScript defines three other useful classes. The Date class defines objects that represent dates. The RegExp class defines objects that represent regular expressions (a powerful pattern-matching tool described inChapter 10). And the Error class defines objects that represent syntax and runtime errors that can occur in a JavaScript program. You can define your own classes of objects by defining appropriate constructor functions. This is explained in Chapter 9.

The JavaScript interpreter performs automatic *garbage collection* for memory management. This means that a program can create objects as needed, and the programmer never needs to worry about destruction or deallocation of those objects. When an object is no longer reachable—when a program no longer has any way to refer to it—the interpreter knows it can never be used again and automatically reclaims the memory it was occupying.

JavaScript is an object-oriented language. Loosely, this means that rather than having globally defined functions to operate on values of various types, the types themselves define *methods* for working with values. To sort the elements of an array `a`, for example, we don't pass `a` to a `sort()` function. Instead, we invoke the `sort()`method of `a`:

```
a.sort();         // The object-oriented version of sort(a).
```

Method definition is covered in Chapter 9. Technically, it is only JavaScript objects that have methods. But numbers, strings, and boolean values behave as if they had methods (Wrapper Objects explains how this works). In JavaScript, `null` and `undefined` are the only values that methods cannot be invoked on.

JavaScript's types can be divided into primitive types and object types. And they can be divided into types with methods and types without. They can also be categorized as *mutable* and *immutable* types. A value of a mutable type can change. Objects and arrays are mutable: a JavaScript program can change the values of object properties and array elements. Numbers, booleans, `null`, and `undefined` are immutable—it doesn't even make sense to talk about changing the value of a number, for example. Strings can be thought of as arrays of characters, and you might expect them to be mutable. In JavaScript, however, strings are immutable: you can access the text at any index of a string, but JavaScript provides no way to alter the text of an existing string. The differences between mutable and immutable values are explored further in Immutable Primitive Values and Mutable Object References.

JavaScript converts values liberally from one type to another. If a program expects a string, for example, and you give it a number, it will automatically convert the number to a string for you. If you use a nonboolean value where a boolean is expected, JavaScript will convert accordingly. The rules for value conversion are explained inType Conversions. JavaScript's liberal value conversion rules affect its definition of equality, and the `==` equality operator performs type conversions as described in Conversions and Equality.

JavaScript variables are *untyped*: you can assign a value of any type to a variable, and you can later assign a value of a different type to the same variable. Variables are *declared* with the `var` keyword. JavaScript uses*lexical scoping*. Variables declared outside of a function are *global variables* and are visible everywhere in a JavaScript program. Variables declared inside a function have *function scope* and are visible only to code that appears inside that function. Variable declaration and scope are covered in Variable Declaration and Variable Scope.

Numbers

Unlike many languages, JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard,[1] which means it can represent numbers as large as $\pm 1.7976931348623157 \times 10^{308}$ and as small as $\pm 5 \times 10^{-324}$.

The JavaScript number format allows you to exactly represent all integers between −9007199254740992 (-2^{53}) and 9007199254740992 (2^{53}), inclusive. If you use integer values larger than this, you may lose precision in the trailing digits. Note, however, that certain operations in JavaScript (such as array indexing and the bitwise operators described in Chapter 4) are performed with 32-bit integers.

When a number appears directly in a JavaScript program, it's called a *numeric literal*. JavaScript supports numeric literals in several formats, as described in the following sections. Note that any numeric literal can be preceded by a minus sign (-) to make the number negative. Technically, however, - is the unary negation operator (see Chapter 4) and is not part of the numeric literal syntax.

Integer Literals

In a JavaScript program, a base-10 integer is written as a sequence of digits. For example:

```
0

3

10000000
```

In addition to base-10 integer literals, JavaScript recognizes hexadecimal (base-16) values. A hexadecimal literal begins with "0x" or "0X", followed by a string of hexadecimal digits. A hexadecimal digit is one of the digits 0 through 9 or the letters a (or A) through f (or F), which represent values 10 through 15. Here are examples of hexadecimal integer literals:

```
0xff   // 15*16 + 15 = 255 (base 10)
0xCAFE911
```

Although the ECMAScript standard does not support them, some implementations of JavaScript allow you to specify integer literals in octal (base-8) format. An octal literal begins with the digit 0 and is followed by a sequence of digits, each between 0 and 7. For example:

```
0377   // 3*64 + 7*8 + 7 = 255 (base 10)
```

Since some implementations support octal literals and some do not, you should never write an integer literal with a leading zero; you cannot know in this case whether an implementation will interpret it as an octal or decimal value. In the strict mode of ECMAScript 5 ("use strict"), octal literals are explicitly forbidden.

Floating-Point Literals

Floating-point literals can have a decimal point; they use the traditional syntax for real numbers. A real value is represented as the integral part of the number, followed by a decimal point and the fractional part of the number.

Floating-point literals may also be represented using exponential notation: a real number followed by the letter e (or E), followed by an optional plus or minus sign, followed by an integer exponent. This notation represents the real number multiplied by 10 to the power of the exponent.

More succinctly, the syntax is:

```
[digits][.digits][(E|e)[(+|-)]digits]
```

For example:

```
3.14

2345.789

.333333333333333333

6.02e23        // 6.02 × 10^23
1.4738223E-32  // 1.4738223 × 10^-32
```

Arithmetic in JavaScript

JavaScript programs work with numbers using the arithmetic operators that the language provides. These include`+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for modulo (remainder after division). Full details on these and other operators can be found in Chapter 4.

In addition to these basic arithmetic operators, JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the `Math` object:

```
Math.pow(2,53)          // => 9007199254740992: 2 to the power 
53

Math.round(.6)          // => 1.0: round to the nearest integer

Math.ceil(.6)           // => 1.0: round up to an integer

Math.floor(.6)          // => 0.0: round down to an integer

Math.abs(-5)            // => 5: absolute value

Math.max(x,y,z)         // Return the largest argument

Math.min(x,y,z)         // Return the smallest argument
Math.random()            // Pseudo-random number x where 0 <= x <
1.0

Math.PI                  // π: circumference of a circle /
diameter

Math.E                   // e: The base of the natural logarithm
Math.sqrt(3)             // The square root of 3
Math.pow(3, 1/3)         // The cube root of 3
Math.sin(0)              // Trigonometry: also Math.cos,
Math.atan, etc.
Math.log(10)             // Natural logarithm of 10
Math.log(100)/Math.LN10  // Base 10 logarithm of 100
Math.log(512)/Math.LN2   // Base 2 logarithm of 512
Math.exp(3)              // Math.E cubed
```

See the Math object in the reference section for complete details on all the mathematical functions supported by JavaScript.

Arithmetic in JavaScript does not raise errors in cases of overflow, underflow, or division by zero. When the result of a numeric operation is larger than the largest representable number (overflow), the result is a special infinity value, which JavaScript prints as `Infinity`. Similarly, when a negative value becomes larger than the largest representable negative number, the result is negative infinity, printed as `-Infinity`. The infinite values behave as you would expect: adding, subtracting, multiplying, or dividing them by anything results in an infinite value (possibly with the sign reversed).

Underflow occurs when the result of a numeric operation is closer to zero than the smallest representable number. In this case, JavaScript returns 0. If underflow occurs from a negative number, JavaScript returns a special value known as "negative zero." This value is almost completely indistinguishable from regular zero and JavaScript programmers rarely need to detect it.

Division by zero is not an error in JavaScript: it simply returns infinity or negative infinity. There is one exception, however: zero divided by zero does not have a well-defined value, and the result of this operation is the special not-a-number value, printed as `NaN`. `NaN` also arises if you attempt to divide infinity by infinity, or take the square root of a negative number or use arithmetic operators with non-numeric operands that cannot be converted to numbers.

JavaScript predefines global variables `Infinity` and `NaN` to hold the positive infinity and not-a-number value. In ECMAScript 3, these are read/write values and can be changed. ECMAScript 5 corrects this and makes the values read-only.
The `Number` object defines alternatives that are read-only even in ECMAScript 3. Here are some examples:

```
Infinity                        // A read/write variable initialized to
Infinity.
Number.POSITIVE_INFINITY     // Same value, read-only.
1/0                             // This is also the same value.
Number.MAX_VALUE + 1          // This also evaluates to Infinity.

Number.NEGATIVE_INFINITY     // These expressions are negative infinity.
-Infinity
-1/0
-Number.MAX_VALUE - 1
```

```
NaN                                // A read/write variable initialized to
NaN.
Number.NaN                        // A read-only property holding the same
value.
0/0                                // Evaluates to NaN.

Number.MIN_VALUE/2              // Underflow: evaluates to 0
-Number.MIN_VALUE/2             // Negative zero
-1/Infinity                       // Also negative 0
-0
```

The not-a-number value has one unusual feature in JavaScript: it does not compare equal to any other value, including itself. This means that you can't write `x == NaN` to determine whether the value of a variable `x` is `NaN`. Instead, you should write `x != x`. That expression will be true if, and only if, x is `NaN`. The function `isNaN()` is similar. It returns `true` if its argument is `NaN`, or if that argument is a non-numeric value such as a string or an object. The related function `isFinite()` returns `true` if its argument is a number other than `NaN`, `Infinity`, or -`Infinity`.

The negative zero value is also somewhat unusual. It compares equal (even using JavaScript's strict equality test) to positive zero, which means that the two values are almost indistinguishable, except when used as a divisor:

```
var zero = 0;           // Regular zero
var negz = -0;          // Negative zero
zero === negz           // => true: zero and negative zero are equal
1/zero === 1/negz      // => false: infinity and -infinity are not equal
```

Binary Floating-Point and Rounding Errors

There are infinitely many real numbers, but only a finite number of them (18437736874454810627, to be exact) can be represented exactly by the JavaScript floating-point format. This means that when you're working with real numbers in JavaScript, the representation of the number will often be an approximation of the actual number.

The IEEE-754 floating-point representation used by JavaScript (and just about every other modern programming language) is a binary representation, which can exactly represent fractions like `1/2`, `1/8`, and `1/1024`. Unfortunately, the fractions we use most commonly (especially when performing financial calculations) are decimal fractions `1/10`, `1/100`, and so on. Binary floating-point representations cannot exactly represent numbers as simple as `0.1`.

JavaScript numbers have plenty of precision and can approximate `0.1` very closely. But the fact that this number cannot be represented exactly can lead to problems. Consider this code:

```
var x = .3 - .2;     // thirty cents minus 20 cents
var y = .2 - .1;     // twenty cents minus 10 cents
x == y                // => false: the two values are not the same!
x == .1               // => false: .3-.2 is not equal to .1
y == .1               // => true: .2-.1 is equal to .1
```

Because of rounding error, the difference between the approximations of .3 and .2 is not exactly the same as the difference between the approximations of .2 and .1. It is important to understand that this problem is not specific to JavaScript: it affects any programming language that uses binary floating-point numbers. Also, note that the values `x` and `y` in the code above are *very* close to each other and to the correct value. The computed values are adequate for almost any purpose: the problem arises when we attempt to compare values for equality.

A future version of JavaScript may support a decimal numeric type that avoids these rounding issues. Until then you might want to perform critical financial calculations using scaled integers. For example, you might manipulate monetary values as integer cents rather than fractional dollars.

Dates and Times

Core JavaScript includes a `Date()` constructor for creating objects that represent dates and times. These Date objects have methods that provide an API for simple date computations. Date objects are not a fundamental type like numbers are. This section presents a quick tutorial on working with dates. Full details can be found in the reference section:

```
var then = new Date(2010, 0, 1);   // The 1st day of the 1st month of 2010
var later = new Date(2010, 0, 1,   // Same day, at 5:10:30pm, local time
                     17, 10, 30);
var now = new Date();             // The current date and time
var elapsed = now - then;        // Date subtraction: interval in
milliseconds

later.getFullYear()              // => 2010
later.getMonth()                  // => 0: zero-based months
later.getDate()                   // => 1: one-based days
later.getDay()                    // => 5: day of week.  0 is Sunday 5 is
Friday.
later.getHours()                  // => 17: 5pm, local time
later.getUTCHours()              // hours in UTC time; depends on timezone

later.toString()                  // => "Fri Jan 01 2010 17:10:30 GMT-0800
(PST)"
later.toUTCString()              // => "Sat, 02 Jan 2010 01:10:30 GMT"
later.toLocaleDateString()      // => "01/01/2010"
later.toLocaleTimeString()      // => "05:10:30 PM"
later.toISOString()              // => "2010-01-02T01:10:30.000Z"; ES5 only
```

[1] This format should be familiar to Java programmers as the format of the `double` type. It is also the `double` format used in almost all modern implementations of C and C++.

Text

A *string* is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character—strings are JavaScript's type for representing text. The *length* of a string is the number of 16-bit values it contains. JavaScript's strings (and its arrays) use zero-based indexing: the first 16-bit value is at position 0, the second at position 1 and so on. The *empty string* is the string of length 0. JavaScript does not have a special type that represents a single element of a string. To represent a single 16-bit value, simply use a string that has a length of 1.

CHARACTERS, CODEPOINTS, AND JAVASCRIPT STRINGS

JavaScript uses the UTF-16 encoding of the Unicode character set, and JavaScript strings are sequences of unsigned 16-bit values. The most commonly used Unicode characters (those from the "basic multilingual plane") have codepoints that fit in 16 bits and can be represented by a single element of a string. Unicode characters whose codepoints do not fit in 16 bits are encoded following the rules of UTF-16 as a sequence (known as a “surrogate pair”) of two 16-bit values. This means that a JavaScript string of length 2 (two 16-bit values) might represent only a single Unicode character:

```
var p = "π"; // π is 1 character with 16-bit codepoint 0x03c0
var e = "e"; // e is 1 character with 17-bit codepoint 0x1d452
p.length     // => 1: p consists of 1 16-bit element
e.length     // => 2: UTF-16 encoding of e is 2 16-bit values:
"\ud835\udc52"
```

The various string-manipulation methods defined by JavaScript operate on 16-bit values, not on characters. They do not treat surrogate pairs specially, perform no normalization of the string, and do not even ensure that a string is well-formed UTF-16.

String Literals

To include a string literally in a JavaScript program, simply enclose the characters of the string within a matched pair of single or double quotes (`'` or `"`). Double-quote characters may be contained within strings delimited by single-quote characters, and single-quote characters may be contained within strings delimited by double quotes. Here are examples of string literals:

```
""   // The empty string: it has zero characters
'testing'
"3.14"
'name="myform"'
"Wouldn't you prefer O'Reilly's book?"
"This string\nhas two lines"
```

```
"π is the ratio of a circle's circumference to its diameter"
```

In ECMAScript 3, string literals must be written on a single line. In ECMAScript 5, however, you can break a string literal across multiple lines by ending each line but the last with a backslash (`\`). Neither the backslash nor the line terminator that follow it are part of the string literal. If you need to include a newline character in a string literal, use the character sequence `\n` (documented below):

```
"two\nlines"    // A string representing 2 lines written on one line
"one\          // A one-line string written on 3 lines. ECMAScript 5 only.
 long\
 line"
```

Note that when you use single quotes to delimit your strings, you must be careful with English contractions and possessives, such as *can't* and *O'Reilly's*. Since the apostrophe is the same as the single-quote character, you must use the backslash character (`\`) to "escape" any apostrophes that appear in single-quoted strings (escapes are explained in the next section).

In client-side JavaScript programming, JavaScript code may contain strings of HTML code, and HTML code may contain strings of JavaScript code. Like JavaScript, HTML uses either single or double quotes to delimit its strings. Thus, when combining JavaScript and HTML, it is a good idea to use one style of quotes for JavaScript and the other style for HTML. In the following example, the string "Thank you" is single-quoted within a JavaScript expression, which is then double-quoted within an HTML event-handler attribute:

```
<button onclick="alert('Thank you')">Click Me</button>
```

Escape Sequences in String Literals

The backslash character (`\`) has a special purpose in JavaScript strings. Combined with the character that follows it, it represents a character that is not otherwise representable within the string. For example, `\n` is an *escape sequence* that represents a newline character.

Another example, mentioned above, is the `\'` escape, which represents the single quote (or apostrophe) character. This escape sequence is useful when you need to include an apostrophe in a string literal that is contained within single quotes. You can see why these are called escape sequences: the backslash allows you to escape from the usual interpretation of the single-quote character. Instead of using it to mark the end of the string, you use it as an apostrophe:

```
'You\'re right, it can\'t be a quote'
```

Table 3-1 lists the JavaScript escape sequences and the characters they represent. Two escape sequences are generic and can be used to represent any character by specifying its Latin-1 or Unicode character code as a hexadecimal number. For example, the sequence `\xA9` represents the copyright symbol, which has the Latin-1 encoding given by the hexadecimal number A9. Similarly, the `\u` escape represents an arbitrary Unicode character specified by four hexadecimal digits; `\u03c0` represents the character π, for example.

Table 3-1. JavaScript escape sequences

Sequence	Character represented
`\0`	The NUL character (`\u0000`)
`\b`	Backspace (`\u0008`)

Sequence	Character represented
`\t`	Horizontal tab (`\u0009`)
`\n`	Newline (`\u000A`)
`\v`	Vertical tab (`\u000B`)
`\f`	Form feed (`\u000C`)
`\r`	Carriage return (`\u000D`)
`\"`	Double quote (`\u0022`)
`\'`	Apostrophe or single quote (`\u0027`)
`\\`	Backslash (`\u005C`)
`\x` *XX*	The Latin-1 character specified by the two hexadecimal digits *XX*
`\u` *XXXX*	The Unicode character specified by the four hexadecimal digits *XXXX*

If the `\` character precedes any character other than those shown in Table 3-1, the backslash is simply ignored (although future versions of the language may, of course, define new escape sequences). For example, `\#` is the same as `#`. Finally, as noted above, ECMAScript 5 allows a backslash before a line break to break a string literal across multiple lines.

Working with Strings

One of the built-in features of JavaScript is the ability to *concatenate* strings. If you use the `+` operator with numbers, it adds them. But if you use this operator on strings, it joins them by appending the second to the first. For example:

```
msg = "Hello, " + "world";   // Produces the string "Hello, world"
greeting = "Welcome to my blog," + " " + name;
```

To determine the length of a string—the number of 16-bit values it contains—use the `length` property of the string. Determine the length of a string `s` like this:

```
s.length
```

In addition to this `length` property, there are a number of methods you can invoke on strings (as always, see the reference section for complete details):

```
var s = "hello, world"          // Start with some text.
s.charAt(0)                      // => "h": the first character.
s.charAt(s.length-1)            // => "d": the last character.
s.substring(1,4)                 // => "ell": the 2nd, 3rd and 4th
characters.
s.slice(1,4)                     // => "ell": same thing
s.slice(-3)                      // => "rld": last 3 characters
s.indexOf("l")                   // => 2: position of first letter l.
s.lastIndexOf("l")              // => 10: position of last letter l.
s.indexOf("l", 3)                // => 3: position of first "l" at or after
3
s.split(", ")                    // => ["hello", "world"] split into
substrings
s.replace("h", "H")             // => "Hello, world": replaces all
instances
s.toUpperCase()                  // => "HELLO, WORLD"
```

Remember that strings are immutable in JavaScript. Methods like `replace()` and `toUpperCase()` return new strings: they do not modify the string on which they are invoked.

In ECMAScript 5, strings can be treated like read-only arrays, and you can access individual characters (16-bit values) from a string using square brackets instead of the `charAt()` method:

```
s = "hello, world";
s[0]                        // => "h"
s[s.length-1]               // => "d"
```

Mozilla-based web browsers such as Firefox have allowed strings to be indexed in this way for a long time. Most modern browsers (with the notable exception of IE) followed Mozilla's lead even before this feature was standardized in ECMAScript 5.

Pattern Matching

JavaScript defines a `RegExp()` constructor for creating objects that represent textual patterns. These patterns are described with *regular expressions*, and JavaScript adopts Perl's syntax for regular expressions. Both strings and RegExp objects have methods for performing pattern matching and search-and-replace operations using regular expressions.

RegExps are not one of the fundamental types of JavaScript. Like Dates, they are simply a specialized kind of object, with a useful API. The regular expression grammar is complex and the API is nontrivial. They are documented in detail in Chapter 10. Because RegExps are powerful and commonly used for text processing, however, this section provides a brief overview.

Although RegExps are not one of the fundamental data types in the language, they do have a literal syntax and can be encoded directly into JavaScript programs. Text between a pair of slashes constitutes a regular expression literal. The second slash in the pair can also be followed by one or more letters, which modify the meaning of the pattern. For example:

```
/^HTML/                 // Match the letters H T M L at the start of a
string
/[1-9][0-9]*/         // Match a non-zero digit, followed by any # of
digits
/\bjavascript\b/i     // Match "javascript" as a word, case-insensitive
```

RegExp objects define a number of useful methods, and strings also have methods that accept RegExp arguments. For example:

```
var text = "testing: 1, 2, 3";    // Sample text
var pattern = /\d+/g               // Matches all instances of one or more
digits
pattern.test(text)                 // => true: a match exists
text.search(pattern)               // => 9: position of first match
text.match(pattern)                // => ["1", "2", "3"]: array of all
matches
text.replace(pattern, "#");        // => "testing: #, #, #"
text.split(/\D+/);                 // => ["","1","2","3"]: split on non-
digits
```

null and undefined

`null` is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value. Using the `typeof` operator on `null` returns the string “object”, indicating that `null` can be thought of as a special object value that indicates “no object”. In practice, however, `null` is typically regarded as the sole member of its own type, and it can be used to indicate "no value" for numbers and strings as well as objects. Most programming languages have an equivalent to JavaScript's `null`: you may be familiar with it as `null` or`nil`.

JavaScript also has a second value that indicates absence of value. The undefined value represents a deeper kind of absence. It is the value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist. The undefined value is also returned by functions that have no return value, and the value of function parameters for which no argument is supplied. `undefined` is a predefined global variable (not a language keyword like `null`) that is initialized to the undefined value. In ECMAScript 3, `undefined` is a read/write variable, and it can be set to any value. This error is corrected in ECMAScript 5 and `undefined` is read-only in that version of the language. If you apply the `typeof` operator to the undefined value, it returns "undefined", indicating that this value is the sole member of a special type.

Despite these differences, `null` and `undefined` both indicate an absence of value and can often be used interchangeably. The equality operator `==` considers them to be equal. (Use the strict equality operator `===` to distinguish them.) Both are falsy values—they behave like `false` when a boolean value is required. Neither`null` nor `undefined` have any properties or methods. In fact, using `.` or `[]` to access a property or method of these values causes a TypeError.

You might consider `undefined` to represent a system-level, unexpected, or error-like absence of value and `null`to represent program-level, normal, or expected absence of value. If you need to assign one of these values to a variable or property or pass one of these values to a function, `null` is almost always the right choice.

The Global Object

The sections above have explained JavaScript’s primitive types and values. Object types—objects, arrays, and functions—are covered in chapters of their own later in this book. But there is one very important object value that we must cover now. The *global object* is a regular JavaScript object that serves a very important purpose: the properties of this object are the globally defined symbols that are available to a JavaScript program. When the JavaScript interpreter starts (or whenever a web browser loads a new page), it creates a new global object and gives it an initial set of properties that define:

- global properties like `undefined`, `Infinity`, and `NaN`
- global functions like `isNaN()`, `parseInt()` (Explicit Conversions), and `eval()` (Evaluation Expressions).
- constructor functions like `Date()`, `RegExp()`, `String()`, `Object()`, and `Array()` (Explicit Conversions)
- global objects like Math and JSON (Serializing Objects)

The initial properties of the global object are not reserved words, but they deserve to be treated as if they are.Reserved Words lists each of these properties. This chapter has already described some of these global properties. Most of the others will be covered elsewhere in this book. And you can look them all up by name in the core JavaScript reference section, or look up the global object itself under the name “Global”. For client-side JavaScript, the Window object defines other globals that you can look up in the client-side reference section.

In top-level code—JavaScript code that is not part of a function—you can use the JavaScript keyword `this` to refer to the global object:

```
var global = this;   // Define a global variable to refer to the global
object
```

In client-side JavaScript, the Window object serves as the global object for all JavaScript code contained in the browser window it represents. This global Window object has a self-referential `window` property that can be used instead of `this` to refer to the global object. The Window object defines the core global properties, but it also defines quite a few other globals that are specific to web browsers and client-side JavaScript.

When first created, the global object defines all of JavaScript's predefined global values. But this special object also holds program-defined globals as well. If your code declares a global variable, that variable is a property of the global object. Variables As Properties explains this in more detail.

Wrapper Objects

JavaScript objects are composite values: they are a collection of properties or named values. We refer to the value of a property using the `.` notation. When the value of a property is a function, we call it a *method*. To invoke the method `m` of an object `o`, we write `o.m()`.

We've also seen that strings have properties and methods:

```
var s = "hello world!";                              // A string
var word = s.substring(s.indexOf(" ")+1, s.length); // Use string
properties
```

Strings are not objects, though, so why do they have properties? Whenever you try to refer to a property of a string `s`, JavaScript converts the string value to an object as if by calling `new String(s)`. This object inherits (see Inheritance) string methods and is used to resolve the property reference. Once the property has been resolved, the newly created object is discarded. (Implementations are not required to actually create and discard this transient object: they must behave as if they do, however.)

Numbers and booleans have methods for the same reason that strings do: a temporary object is created using the`Number()` or `Boolean()` constructor, and the method is resolved using that temporary object. There are not wrapper objects for the `null` and `undefined` values: any attempt to access a property of one of these values causes a `TypeError`.

Consider the following code and think about what happens when it is executed:

```
var s = "test";           // Start with a string value.
s.len = 4;                 // Set a property on it.
var t = s.len;            // Now query the property.
```

When you run this code, the value of `t` is `undefined`. The second line of code creates a temporary String object, sets its `len` property to 4, and then discards that object. The third line creates a new String object from the original (unmodified) string value and then tries to read the `len` property. This property does not exist, and the expression evaluates to `undefined`. This code demonstrates that strings, numbers, and boolean values behave like objects when you try to read the value of a property (or method) from them. But if you attempt to set the value of a property, that attempt is silently ignored: the change is made on a temporary object and does not persist.

The temporary objects created when you access a property of a string, number, or boolean are known as *wrapper objects*, and it may occasionally be necessary to distinguish a string value from a String object or a number or boolean value from a Number or Boolean object. Usually, however, wrapper objects can be considered an implementation detail and you don’t have to think about them. You just need to know that string, number, and boolean values differ from objects in that their properties are read-only and that you can't define new properties on them.

Note that it is possible (but almost never necessary or useful) to explicitly create wrapper objects, by invoking the`String()`, `Number()`, or `Boolean()` constructors:

```
var s = "test", n = 1, b = true;   // A string, number, and boolean value.
var S = new String(s);              // A String object
var N = new Number(n);              // A Number object
var B = new Boolean(b);             // A Boolean object
```

JavaScript converts wrapper objects into the wrapped primitive value as necessary, so the objects `S`, `N`, and `B`above usually, but not always, behave just like the values `s`, `n`, and `b`. The `==` equality operator treats a value and its wrapper object as equal, but you can distinguish them with the `===` strict equality operator.
The `typeof`operator will also show you the difference between a primitive value and its wrapper object.

Immutable Primitive Values and Mutable Object References

There is a fundamental difference in JavaScript between primitive values (`undefined`, `null`, booleans, numbers, and strings) and objects (including arrays and functions). Primitives are immutable: there is no way to change (or "mutate") a primitive value. This is obvious for numbers and booleans—it doesn't even make sense to change the value of a number. It is not so obvious for strings, however. Since strings are like arrays of characters, you might expect to be able to alter the character at any specified index. In fact, JavaScript does not allow this, and all string methods that appear to return a modified string are, in fact, returning a new string value. For example:

```
var s = "hello";    // Start with some lowercase text
s.toUpperCase();    // Returns "HELLO", but doesn't alter s
```

```
s                        // => "hello": the original string has not changed
```

Primitives are also compared *by value*: two values are the same only if they have the same value. This sounds circular for numbers, booleans, `null`, and `undefined`: there is no other way that they could be compared. Again, however, it is not so obvious for strings. If two distinct string values are compared, JavaScript treats them as equal if, and only if, they have the same length and if the character at each index is the same.

Objects are different than primitives. First, they are *mutable*—their values can change:

```
var o = { x:1 };       // Start with an object
o.x = 2;                // Mutate it by changing the value of a property
o.y = 3;                // Mutate it again by adding a new property

var a = [1,2,3]        // Arrays are also mutable
a[0] = 0;               // Change the value of an array element
a[3] = 4;               // Add a new array element
```

Objects are not compared by value: two objects are not equal even if they have the same properties and values. And two arrays are not equal even if they have the same elements in the same order:

```
var o = {x:1}, p = {x:1};  // Two objects with the same properties
o === p                      // => false: distinct objects are never equal
var a = [], b = [];        // Two distinct, empty arrays
a === b                      // => false: distinct arrays are never equal
```

Objects are sometimes called *reference types* to distinguish them from JavaScript's primitive types. Using this terminology, object values are *references*, and we say that objects are compared *by reference*: two object values are the same if and only if they *refer* to the same underlying object.

```
var a = [];    // The variable a refers to an empty array.
var b = a;     // Now b refers to the same array.
b[0] = 1;      // Mutate the array referred to by variable b.
a[0]            // => 1: the change is also visible through variable a.
```

```
a === b         // => true: a and b refer to the same object, so they are
equal.
```

As you can see from the code above, assigning an object (or array) to a variable simply assigns the reference: it does not create a new copy of the object. If you want to make a new copy of an object or array, you must explicitly copy the properties of the object or the elements of the array. This example demonstrates using a `for`loop (for):

```
var a = ['a','b','c'];                  // An array we want to copy
var b = [];                              // A distinct array we'll copy into
for(var i = 0; i < a.length; i++) { // For each index of a[]
     b[i] = a[i];                        // Copy an element of a into b
}
```

Similarly, if we want to compare two distinct objects or arrays, we must compare their properties or elements. This code defines a function to compare two arrays:

```
function equalArrays(a,b) {
     if (a.length != b.length) return false; // Different-size arrays not
equal
     for(var i = 0; i < a.length; i++)        // Loop through all elements
          if (a[i] !== b[i]) return false;     // If any differ, arrays not
equal
     return true;                               // Otherwise they are
equal
}
```

Type Conversions

JavaScript is very flexible about the types of values it requires. We’ve seen this for booleans: when JavaScript expects a boolean value, you may supply a value of any type, and JavaScript will convert it as needed. Some values (“truthy” values) convert to `true` and others (“falsy” values) convert to `false`. The same is true for other types: if JavaScript wants a string, it will convert whatever value you give it to a string. If JavaScript wants a number, it will try to convert the value you give it to a number (or to `NaN` if it cannot perform a meaningful conversion). Some examples:

```
10 + " objects"      // => "10 objects".  Number 10 converts to a string
"7" * "4"             // => 28: both strings convert to numbers
var n = 1 - "x";     // => NaN: string "x" can't convert to a number
n + " objects"       // => "NaN objects": NaN converts to string "NaN"
```

Table 3-2 summarizes how values convert from one type to another in JavaScript. Bold entries in the table highlight conversions that you may find surprising. Empty cells indicate that no conversion is necessary and none is performed.

Table 3-2. JavaScript type conversions

Value	**Converted to:**			
	String	**Number**	**Boolean**	**Object**
`undefined`	`"undefined"`	`NaN`	`false`	*throws T*
`null`	`"null"`	**`0`**	`false`	*throws T*
`true`	`"true"`	**`1`**		`new Boole`
`false`	`"false"`	**`0`**		`new Boole`
`""` (empty string)		**`0`**	**`false`**	`new Stri`
`"1.2"` (nonempty, numeric)		`1.2`	`true`	`new Stri`
`"one"` (nonempty, non-numeric)		`NaN`	`true`	`new Stri`
`0`	`"0"`		**`false`**	`new Numbe`
`-0`	`"0"`		**`false`**	`new Numbe`
`NaN`	`"NaN"`		**`false`**	`new Numbe`
`Infinity`	`"Infinity"`		`true`	`new Number(I`
`-Infinity`	`"-Infinity"`		`true`	`new Numbe Infinity)`

Value	**Converted to:**			
	String	**Number**	**Boolean**	**Object**
`1` (finite, non-zero)	`"1"`		`true`	`new Numbe`
`{}` (any object)	*see Object to Primitive Conversions*	*see Object to Primitive Conversions*	`true`	
`[]` (empty array)	`""`	`0`	`true`	
`[9]` (1 numeric elt)	`"9"`	`9`	`true`	
`['a']` (any other array)	*use join() method*	`NaN`	`true`	
`function(){}` (any function)	*see Object to Primitive Conversions*	`NaN`	`true`	

The primitive-to-primitive conversions shown in the table are relatively straightforward. Conversion to boolean was already discussed in Boolean Values. Conversion to strings is well-defined for all primitive values. Conversion to numbers is just a little trickier. Strings that can be parsed as numbers convert to those numbers. Leading and trailing spaces are allowed, but any leading or trailing nonspace characters that are not part of a numeric literal cause the string-to-number conversion to produce `NaN`. Some numeric conversions may seem surprising: `true` converts to 1, and `false` and the empty string `""` convert to 0.

Primitive-to-object conversions are straightforward: primitive values convert to their wrapper object (Wrapper Objects) as if by calling the `String()`, `Number()`, or `Boolean()` constructor. The exceptions are `null` and`undefined`: any attempt to use these values where an object is expected raises a TypeError exception rather than performing a conversion.

Object-to-primitive conversion is somewhat more complicated, and it is the subject of Object to Primitive Conversions.

Conversions and Equality

Because JavaScript can convert values flexibly, its `==` equality operator is also flexible with its notion of equality. All of the following comparisons are true, for example:

```
null == undefined // These two values are treated as equal.
"0" == 0            // String converts to a number before comparing.
0 == false          // Boolean converts to number before comparing.
"0" == false        // Both operands convert to numbers before comparing.
```

Equality and Inequality Operators explains exactly what conversions are performed by the `==` operator in order to determine whether two values should be considered equal, and it also describes the strict equality operator `===`that does not perform conversions when testing for equality.

Keep in mind that convertibility of one value to another does not imply equality of those two values. If`undefined` is used where a boolean value is expected, for example, it will convert to `false`. But this does not mean that `undefined == false`. JavaScript operators and statements expect values of various types, and perform conversions to those types. The `if` statement converts `undefined` to `false`, but the `==` operator never attempts to convert its operands to booleans.

Explicit Conversions

Although JavaScript performs many type conversions automatically, you may sometimes need to perform an explicit conversion, or you may prefer to make the conversions explicit to keep your code clearer.

The simplest way to perform an explicit type conversion is to use the `Boolean()`, `Number()`, `String()`, or`Object()` functions. We've already seen these functions as constructors for wrapper objects (in Wrapper Objects). When invoked without the `new` operator, however, they work as conversion functions and perform the conversions summarized in Table 3-2:

```
Number("3")              // => 3
String(false)            // => "false"  Or use false.toString()
Boolean([])              // => true
Object(3)                // => new Number(3)
```

Note that any value other than `null` or `undefined` has a `toString()` method and the result of this method is usually the same as that returned by the `String()` function. Also note that Table 3-2 shows a TypeError if you attempt to convert `null` or `undefined` to an object. The `Object()` function does not throw an exception in this case: instead it simply returns a newly created empty object.

Certain JavaScript operators perform implicit type conversions, and are sometimes used for the purposes of type conversion. If one operand of the `+` operator is a string, it converts the other one to a string. The unary `+` operator converts its operand to a number. And the unary `!` operator converts its operand to a boolean and negates it. These facts lead to the following type conversion idioms that you may see in some code:

```
x + ""               // Same as String(x)
+x                   // Same as Number(x).  You may also see x-0
!!x                  // Same as Boolean(x). Note double !
```

Formatting and parsing numbers are common tasks in computer programs and JavaScript has specialized functions and methods that provide more precise control over number-to-string and string-to-number conversions.

The `toString()` method defined by the Number class accepts an optional argument that specifies a radix, or base, for the conversion. If you do not specify the argument, the conversion is done in base 10. However, you can also convert numbers in other bases (between 2 and 36). For example:

```
var n = 17;
binary_string = n.toString(2);          // Evaluates to "10001"
octal_string = "0" + n.toString(8);    // Evaluates to "021"
hex_string = "0x" + n.toString(16);    // Evaluates to "0x11"
```

When working with financial or scientific data, you may want to convert numbers to strings in ways that give you control over the number of decimal places or the number of significant digits in the output, or you may want to control whether exponential notation is used. The Number class defines three methods for these kinds of number-to-string conversions. `toFixed()` converts a number to a string with a specified number of digits after the decimal point. It never uses exponential notation. `toExponential()` converts a number to a string using exponential notation, with one digit before the decimal point and a specified number of digits after the decimal point (which means that the number of significant digits is one larger than the value you specify).`toPrecision()` converts a number to a string with the number of significant digits you specify. It uses exponential notation if the number of significant digits is not large enough to display the entire integer portion of the number. Note that all three methods round the trailing digits or pad with zeros as appropriate. Consider the following examples:

```
var n = 123456.789;
n.toFixed(0);           // "123457"
n.toFixed(2);           // "123456.79"
n.toFixed(5);           // "123456.78900"
n.toExponential(1);    // "1.2e+5"
n.toExponential(3);    // "1.235e+5"
n.toPrecision(4);      // "1.235e+5"
n.toPrecision(7);      // "123456.8"
n.toPrecision(10);     // "123456.7890"
```

If you pass a string to the `Number()` conversion function, it attempts to parse that string as an integer or floating-point literal. That function only works for base-10 integers, and does not allow trailing characters that are not part of the literal. The `parseInt()` and `parseFloat()` functions (these are global functions, not methods of any class) are more flexible. `parseInt()` parses only integers, while `parseFloat()` parses both integers and floating-point numbers. If a string begins with “0x” or “0X”, `parseInt()` interprets it as a hexadecimal number.[2] Both `parseInt()` and `parseFloat()` skip leading whitespace, parse as many numeric characters as they can, and ignore anything that follows. If the first nonspace character is not part of a valid numeric literal, they return NaN:

```
parseInt("3 blind mice")      // => 3
parseFloat(" 3.14 meters")    // => 3.14
parseInt("-12.34")             // => -12
parseInt("0xFF")               // => 255
parseInt("0xff")               // => 255
parseInt("-0XFF")              // => -255
parseFloat(".1")               // => 0.1
```

```
parseInt("0.1")                // => 0
parseInt(".1")                 // => NaN: integers can't start with "."
parseFloat("$72.47");          // => NaN: numbers can't start with "$"
```

`parseInt()` accepts an optional second argument specifying the radix (base) of the number to be parsed. Legal values are between 2 and 36. For example:

```
parseInt("11", 2);             // => 3 (1*2 + 1)
parseInt("ff", 16);            // => 255 (15*16 + 15)
parseInt("zz", 36);            // => 1295 (35*36 + 35)
parseInt("077", 8);            // => 63 (7*8 + 7)
parseInt("077", 10);           // => 77 (7*10 + 7)
```

Object to Primitive Conversions

Object-to-boolean conversions are trivial: all objects (including arrays and functions) convert to `true`. This is so even for wrapper objects: `new Boolean(false)` is an object rather than a primitive value, and so it converts to`true`.

Object-to-string and object-to-number conversions are performed by invoking a method of the object to be converted. This is complicated by the fact that JavaScript objects have two different methods that perform conversions, and it is also complicated by some special cases described below. Note that the string and number conversion rules described here apply only to native objects. Host objects (defined by web browsers, for example) can convert to numbers and strings according to their own algorithms.

All objects inherit two conversion methods. The first is called `toString()`, and its job is to return a string representation of the object. The default `toString()` method does not return a very interesting value (though we’ll find it useful in Example 6-4):

```
({x:1, y:2}).toString()     // => "[object Object]"
```

Many classes define more specific versions of the `toString()` method. The `toString()` method of the Array class, for example, converts each array element to a string and joins the resulting strings together with commas in between. The `toString()` method of the Function class returns an implementation-defined representation of a function. In practice, implementations usually convert user-defined functions to strings of JavaScript source code. The Date class defines a `toString()` method that returns a human-readable (and JavaScript-parsable) date and time string. The RegExp class defines a `toString()` method that converts RegExp objects to a string that looks like a RegExp literal:

```
[1,2,3].toString()                    // => "1,2,3"
(function(x) { f(x); }).toString()  // => "function(x) {\n    f(x);\n}"
/\d+/g.toString()                    // => "/\\d+/g"
new Date(2010,0,1).toString()   // => "Fri Jan 01 2010 00:00:00 GMT-0800
(PST)"
```

The other object conversion function is called `valueOf()`. The job of this method is less well-defined: it is supposed to convert an object to a primitive value that represents the object, if any such primitive value exists. Objects are compound values, and most objects cannot really be represented by a single primitive value, so the default `valueOf()` method simply returns the object itself rather than returning a primitive. Wrapper classes define `valueOf()` methods that return the wrapped primitive value. Arrays, functions, and regular expressions simply inherit the default method. Calling `valueOf()` for instances of these types simply returns the object itself. The Date class defines a `valueOf()` method that returns the date in its internal representation: the number of milliseconds since January 1, 1970:

```
var d = new Date(2010, 0, 1);    // January 1st, 2010, (Pacific time)
d.valueOf()                       // => 1262332800000
```

With the `toString()` and `valueOf()` methods explained, we can now cover object-to-string and object-to-number conversions. Do note, however, that there are some special cases in which JavaScript performs a different object-to-primitive conversion. These special cases are covered at the end of this section.

To convert an object to a string, JavaScript takes these steps:

- If the object has a `toString()` method, JavaScript calls it. If it returns a primitive value, JavaScript converts that value to a string (if it is not already a string) and returns the result of that conversion. Note that primitive-to-string conversions are all well-defined in Table 3-2.
- If the object has no `toString()` method, or if that method does not return a primitive value, then JavaScript looks for a `valueOf()` method. If the method exists, JavaScript calls it. If the return value is a primitive, JavaScript converts that value to a string (if it is not already) and returns the converted value.
- Otherwise, JavaScript cannot obtain a primitive value from either `toString()` or `valueOf()`, so it throws a TypeError.

To convert an object to a number, JavaScript does the same thing, but it tries the `valueOf()` method first:

- If the object has a `valueOf()` method that returns a primitive value, JavaScript converts (if necessary) that primitive value to a number and returns the result.
- Otherwise, if the object has a `toString()` method that returns a primitive value, JavaScript converts and returns the value.
- Otherwise, JavaScript throws a TypeError.

The details of this object-to-number conversion explain why an empty array converts to the number 0 and why an array with a single element may also convert to a number. Arrays inherit the default `valueOf()` method that returns an object rather than a primitive value, so array-to-number conversion relies on the `toString()` method. Empty arrays convert to the empty string. And the empty string converts to the number 0. An array with a single element converts to the same string that that one element does. If an array contains a single number, that number is converted to a string, and then back to a number.

The `+` operator in JavaScript performs numeric addition and string concatenation. If either of its operands is an object, JavaScript converts the object using a special object-to-primitive conversion rather than the object-to-number conversion used by the other arithmetic operators. The `==` equality operator is similar. If asked to compare an object with a primitive value, it converts the object using the object-to-primitive conversion.

The object-to-primitive conversion used by `+` and `==` includes a special case for Date objects. The Date class is the only predefined core JavaScript type that defines meaningful conversions to both strings and numbers. The object-to-primitive conversion is basically an object-to-number conversion (`valueof()` first) for all objects that are not dates, and an object-to-string conversion (`toString()` first) for Date objects. The conversion is not exactly the same as those explained above, however: the primitive value returned by `valueOf()` or`toString()` is used directly without being forced to a number or string.

The `<` operator and the other relational operators perform object-to-primitive conversions like `==` does, but without the special case for Date objects: any object is converted by trying `valueOf()` first and then`toString()`. Whatever primitive value is obtained is used directly, without being further converted to a number or string.

`+`, `==`, `!=` and the relational operators are the only ones that perform this special kind of string-to-primitive conversions. Other operators convert more explicitly to a specified type and do not have any special case for Date objects. The `-` operator, for example, converts its operands to numbers. The following code demonstrates the behavior of `+`, `-`, `==`, and `>` with Date objects:

```
var now = new Date();       // Create a Date object
typeof (now + 1)             // => "string": + converts dates to strings
typeof (now - 1)             // => "number": - uses object-to-number
conversion
now == now.toString()       // => true: implicit and explicit string
conversions
now > (now -1)               // => true: > converts a Date to a number
```

[2] In ECMAScript 3, `parseInt()` may parse a string that begins with “0” (but not “0x” or “0X”) as an octal number or as a decimal number. Because the behavior is unspecified, you should never use `parseInt()` to parse numbers with leading zeros, unless you explicitly specify the radix to be used! In ECMAScript 5, `parseInt()` only parses octal numbers if you explicitly pass 8 as the second argument.

Variable Declaration

Before you use a variable in a JavaScript program, you should *declare* it. Variables are declared with the `var`keyword, like this:

```
var i;
var sum;
```

You can also declare multiple variables with the same `var` keyword:

```
var i, sum;
```

And you can combine variable declaration with variable initialization:

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

If you don't specify an initial value for a variable with the `var` statement, the variable is declared, but its value is`undefined` until your code stores a value into it.

Note that the `var` statement can also appear as part of the `for` and `for/in` loops (introduced in Chapter 5), allowing you to succinctly declare the loop variable as part of the loop syntax itself. For example:

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var p in o) console.log(p);
```

If you're used to statically typed languages such as C or Java, you will have noticed that there is no type associated with JavaScript's variable declarations. A JavaScript variable can hold a value of any type. For example, it is perfectly legal in JavaScript to assign a number to a variable and then later assign a string to that variable:

```
var i = 10;
i = "ten";
```

Repeated and Omitted Declarations

It is legal and harmless to declare a variable more than once with the `var` statement. If the repeated declaration has an initializer, it acts as if it were simply an assignment statement.

If you attempt to read the value of an undeclared variable, JavaScript generates an error. In ECMAScript 5 strict mode (“use strict”), it is also an error to assign a value to an undeclared variable. Historically, however, and in non-strict mode, if you assign a value to an undeclared variable, JavaScript actually creates that variable as a property of the global object, and it works much like (but not exactly the same as, see Variables As Properties) a properly declared global variable. This means that you can get away with leaving your global variables undeclared. This is a bad habit and a source of bugs, however, and you should always declare your variables with`var`.

Variable Scope

The *scope* of a variable is the region of your program source code in which it is defined. A *global* variable has global scope; it is defined everywhere in your JavaScript code. On the other hand, variables declared within a function are defined only within the body of the function. They are *local* variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable:

```
var scope = "global";           // Declare a global variable
function checkscope() {
     var scope = "local";        // Declare a local variable with the same
name
     return scope;               // Return the local value, not the global
one
}
checkscope()                     // => "local"
```

Although you can get away with not using the `var` statement when you write code in the global scope, you must always use `var` to declare local variables. Consider what happens if you don't:

```
scope = "global";               // Declare a global variable, even without
var.
function checkscope2() {
     scope = "local";           // Oops! We just changed the global
variable.
     myscope = "local";         // This implicitly declares a new global
variable.
     return [scope, myscope]; // Return two values.
}
checkscope2()                   // => ["local", "local"]: has side effects!
scope                            // => "local": global variable has
changed.
myscope                          // => "local": global namespace cluttered
up.
```

Function definitions can be nested. Each function has its own local scope, so it is possible to have several nested layers of local scope. For example:

```
var scope = "global scope";            // A global variable
function checkscope() {
     var scope = "local scope";        // A local variable
     function nested() {
          var scope = "nested scope";  // A nested scope of local variables
          return scope;                  // Return the value in scope here
     }
     return nested();
}
checkscope()                             // => "nested scope"
```

Function Scope and Hoisting

In some C-like programming languages, each block of code within curly braces has its own scope, and variables are not visible outside of the block in which they are declared. This is called *block scope*, and JavaScript does *not*have it. Instead, JavaScript uses *function scope*: variables are visible within the function in which they are defined and within any functions that are nested within that function.

In the following code, the variables `i`, `j`, and `k` are declared in different spots, but all have the same scope—all three are defined throughout the body of the function:

```
function test(o) {
    var i = 0;                       // i is defined throughout function
    if (typeof o == "object") {
        var j = 0;                   // j is defined everywhere, not
just block
        for(var k=0; k < 10; k++) { // k is defined everywhere, not just
loop
            console.log(k);          // print numbers 0 through 9
        }
        console.log(k);              // k is still defined: prints 10
    }
    console.log(j);                  // j is defined, but may not be
initialized
}
```

JavaScript's function scope means that all variables declared within a function are visible *throughout* the body of the function. Curiously, this means that variables are even visible before they are declared. This feature of JavaScript is informally known as *hoisting*: JavaScript code behaves as if all variable declarations in a function (but not any associated assignments) are "hoisted" to the top of the function. Consider the following code:

```
var scope = "global";
```

```
function f() {
     console.log(scope);   // Prints "undefined", not "global"
     var scope = "local"; // Variable initialized here, but defined
everywhere
     console.log(scope);   // Prints "local"
}
```

You might think that the first line of the function would print “global”, because the `var` statement declaring the local variable has not yet been executed. Because of the rules of function scope, however, this is not what happens. The local variable is defined throughout the body of the function, which means the global variable by the same name is hidden throughout the function. Although the local variable is defined throughout, it is not actually initialized until the `var` statement is executed. Thus, the function above is equivalent to the following, in which the variable declaration is “hoisted” to the top and the variable initialization is left where it is:

```
function f() {
     var scope;            // Local variable is declared at the top of the
function
     console.log(scope); // It exists here, but still has "undefined" value
     scope = "local";     // Now we initialize it and give it a value
     console.log(scope); // And here it has the value we expect
}
```

In programming languages with block scope, it is generally good programming practice to declare variables as close as possible to where they are used and with the narrowest possible scope. Since JavaScript does not have block scope, some programmers make a point of declaring all their variables at the top of the function, rather than trying to declare them closer to the point at which they are used. This technique makes their source code accurately reflect the true scope of the variables.

Variables As Properties

When you declare a global JavaScript variable, what you are actually doing is defining a property of the global object (The Global Object). If you use `var` to declare the variable, the property that is created is nonconfigurable (see Property Attributes), which means that it cannot be deleted with the `delete` operator. We've already noted that if you're not using strict mode and you assign a value to an undeclared variable, JavaScript automatically creates a global variable for you. Variables created in this way are regular, configurable properties of the global object and they can be deleted:

```
var truevar = 1;      // A properly declared global variable, nondeletable.
fakevar = 2;           // Creates a deletable property of the global
object.
this.fakevar2 = 3;    // This does the same thing.
delete truevar        // => false: variable not deleted
delete fakevar        // => true: variable deleted
delete this.fakevar2 // => true: variable deleted
```

JavaScript global variables are properties of the global object, and this is mandated by the ECMAScript specification. There is no such requirement for local variables, but you can imagine local variables as the properties of an object associated with each function invocation. The ECMAScript 3 specification referred to this object as the "call object," and the ECMAScript 5 specification calls it a "declarative environment record." JavaScript allows us to refer to the global object with the `this` keyword, but it does not give us any way to refer to the object in which local variables are stored. The precise nature of these objects that hold local variables is an implementation detail that need not concern us. The notion that these local variable objects exist, however, is an important one, and it is developed further in the next section.

The Scope Chain

JavaScript is a *lexically scoped* language: the scope of a variable can be thought of as the set of source code lines for which the variable is defined. Global variables are defined throughout the program. Local variables are defined throughout the function in which they are declared, and also within any functions nested within that function.

If we think of local variables as properties of some kind of implementation-defined object, then there is another way to think about variable scope. Every chunk of JavaScript code (global code or functions) has a *scope chain*associated with it. This scope chain is a list or chain of objects that defines the variables that are “in scope” for that code. When JavaScript needs to look up the value of a variable `x` (a process called *variable resolution*), it starts by looking at the first object in the chain. If that object has a property named `x`, the value of that property is used. If the first object does not have a property named `x`, JavaScript continues the search with the next object in the chain. If the second object does not have a property named `x`, the search moves on to the next object, and so on. If `x` is not a property of any of the objects in the scope chain, then `x` is not in scope for that code, and a ReferenceError occurs.

In top-level JavaScript code (i.e., code not contained within any function definitions), the scope chain consists of a single object, the global object. In a non-nested function, the scope chain consists of two objects. The first is the object that defines the function’s parameters and local variables, and the second is the global object. In a nested function, the scope chain has three or more objects. It is important to understand how this chain of objects is created. When a function is defined, it stores the scope chain then in effect. When that function is invoked, it creates a new object to store its local variables, and adds that new object to the stored scope chain to create a new, longer, chain that represents the scope for that function invocation. This becomes more interesting for nested functions because each time the outer function is called, the inner function is defined again. Since the scope chain differs on each invocation of the outer function, the inner function will be subtly different each time it is defined—the code of the inner function will be identical on each invocation of the outer function, but the scope chain associated with that code will be different.

This notion of a scope chain is helpful for understanding the `with` statement (with) and is crucial for understanding closures (Closures).