MODULE-3 : JavaScript: Client-Side Scripting

What is javascript and What Can it Do?

Larry Ullman, in his *Modern JavaScript: Develop and Design* has an especially succinct definition of JavaScript: it is an object-oriented, dynamically typed, scripting language. In the context of this book, we can add as well, that it is primarily a client-side scripting language

Although it contains the word *Java*, JavaScript and Java are vastly different programming languages with different uses. Java is a full-fledged compiled, object- oriented language, popular for its ability to run on any platform with a Java Virtual Machine installed. Conversely, JavaScript is one of the world's most popular lan- guages, with fewer of the object-oriented features of Java, and runs directly inside the browser, without the need for the JVM.

JavaScript is object oriented in that almost everything in the language is an object. For instance, variables are objects in that they have constructors, properties, and methods . Unlike more familiar object- oriented languages such as Java, C#, and Visual Basic, functions in JavaScript are also objects. T he objects in JavaScript are pro- totype-based rather than class-based, which means that while JavaScript shares some syntactic features of Java or C#, it is also quite different from those languages. Given that JavaScript approaches objects far differently than other languages, and does not have a formal class mechanism nor inheritance syntax, we might say that it is a *strange* object-oriented language.

JavaScript is dynamically typed (also called weakly typed) in that variables can be easily (or implicitly) converted from one data type to another. In a programming language such as Java, variables are statically typed, in that the data type of a vari- able is defined by the programmer (e.g., `int` abc) and enforced by the compiler. With JavaScript, the type of data a variable can hold is assigned at runtime and can change during run time as well.

The final term in the above definition of JavaScript is that it is a client-side scripting language, and due to the importance of this aspect, it will be covered in a bit more detail below.

3.1 Client-side scripting

The idea of **client-side scripting** is an important one in web development. It refers to the client machine (i.e., the browser) running code locally rather than relying on the server to execute code and return the result. There are many client-side lan- guages that have come into use over the past decade including Flash, VBScript, Java, and JavaScript. Some of these technologies only work in certain browsers, while others require plug-ins to function. We will focus on JavaScript due to its browser interoperability (that is, its ability to work/operate on most browsers). Figure 6.1 illustrates how a client machine downloads and executes JavaScript code.

There are many **advantages** of client-side scripting:

- Processing can be offloaded from the server to client machines, thereby reducing the load on the server.
- The browser can respond more rapidly to user events than a request to a remote server ever could, which improves the user experience.

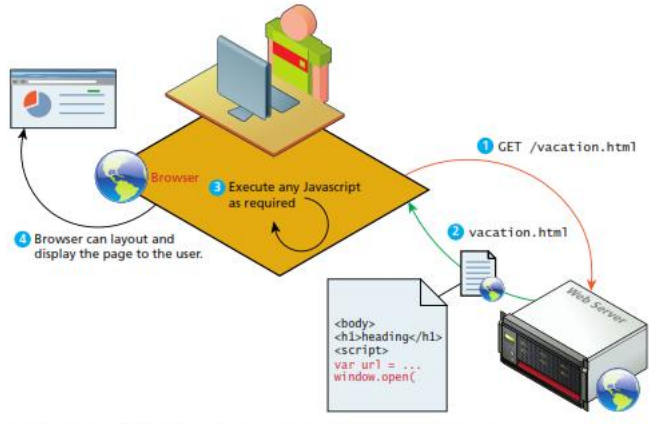

FIGURE 6.1 Downloading and executing a client-side JavaScript script

- JavaScript can interact with the downloaded HTML in a way that the server cannot, creating a user experience more like desktop software than simple HTML ever could.

The disadvantages of client-side scripting are mostly related to how programmers use JavaScript in their applications. Some of these include:

- There is no guarantee that the client has JavaScript enabled, meaning any required functionality must be housed on the server, despite the possibility that it could be offloaded.
- The idiosyncrasies between various browsers and operating systems make it difficult to test for all potential client configurations. What works in one browser, may generate an error in another.
- JavaScript-heavy web applications can be complicated to debug and maintain.

JavaScript has often been used through inline HTML hooks that are embedded into the HTML of a web page. Although this technique has been used for years, it has the distinct disadvantage of blending HTML and JavaScript together, which decreases code readability, and increases the difficulty of web development.

Despite these limitations, the ability to enhance the visual appearance of a web application while potentially reducing the demands on the server make client-side scripting something that is a required competency for the web developer. An under- standing of the concepts will help you avoid JavaScript's pitfalls and allow you to create compelling web applications.

We should mention that JavaScript is not the only type of client-side scripting. There are two other noteworthy client-side approaches to web programming. Perhaps the most familiar of these alternatives is **Adobe Flash**, which is a vector- based drawing and animation program, a video file format, and a software platform that has its own JavaScript-like programming language called **ActionScript**. Flash is often used for animated advertisements and online games, and can also be used to construct web interfaces.

It is worth understanding how Flash works in the browser. Flash objects (not videos) are in a format called SWF (Shockwave Flash) and

are included within an HTML document via the <object> tag. The SWF file is then downloaded by the browser and then the browser delegates control to a plug-in to execute the Flash file, as shown in Figure 6.2. A **browser plug-in** is a software add-on that extends the functionality and capabilities of the browser by allowing it to view and process dif- ferent types of web content.

It should be noted that a browser plug-in is different than a **browser extension**— these also extend the functionality of a browser but are not used to process down- loaded content. For instance, FireBug in the Firefox browser provides a wide range of tools that help the developer understand what's in a page; it doesn't really alter how the browser displays a page.

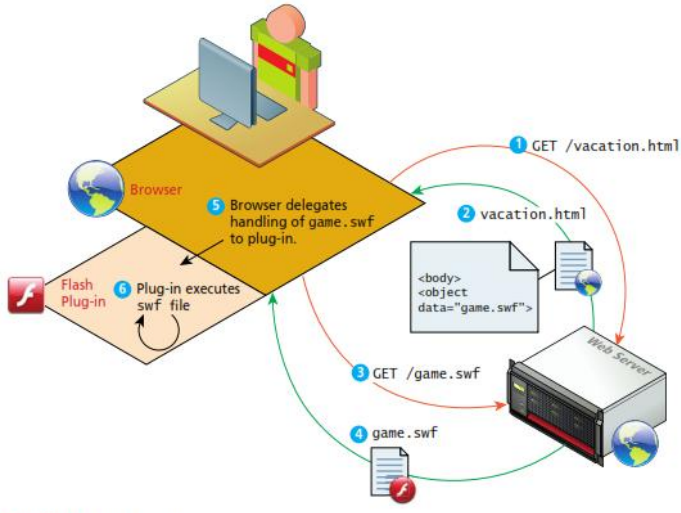

FIGURE 6.2 Adobe Flash

The second (and oldest) of these alternatives to JavaScript is **Java applets**. An **applet** is a term that refers to a small application that performs a relatively small task. Java applets are written using the Java programming language and are sepa- rate objects that are included within an HTML document via the <applet> tag, downloaded, and then passed on to a Java plug-in. This plug-in then passes on the execution of the applet outside the browser to the Java Runtime Environment (JRE) that is installed on the client's machine. Figure 6.3 illustrates how Java applets work in the web environment.

Both Flash plug-ins and Java applets are losing support by major players for a number of reasons. First, Java applets require the JVM be installed and up to date, which some players are not allowing for security reasons (Apple's iOS pow- ering iPhones and iPads supports neither Flash nor Java applets). Second, Flash and Java applets also require frequent updates, which can annoy the user and present security risks. With the universal adoption of JavaScript and HTML5, JavaScript remains the most dynamic and important client-side scripting language for the modern web developer.

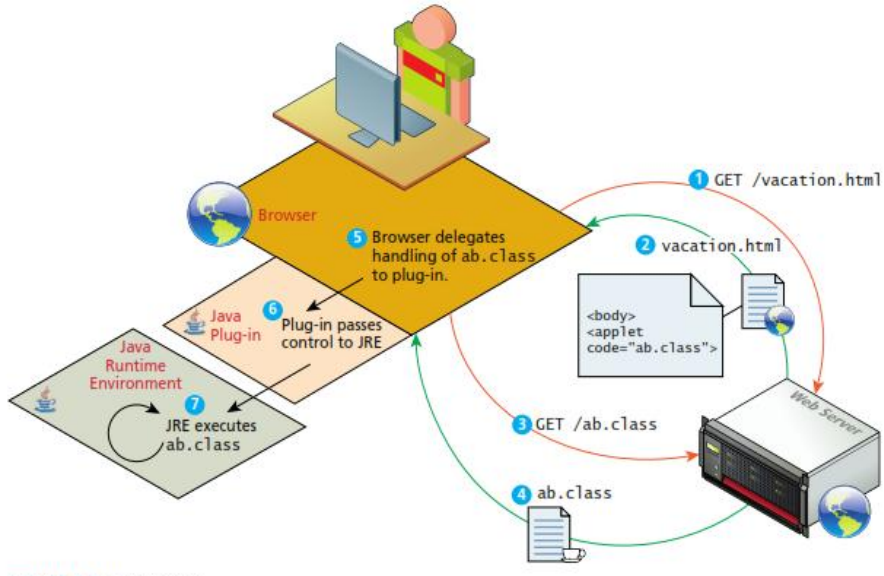

FIGURE 6.3 Java applets

3.1.1 Javascript's history and Uses

JavaScript was introduced by Netscape in their Navigator browser back in 1996. It originally was called LiveScript, but was renamed partly because one of its original purposes was to provide a measure of control within the browser over Java applets. JavaScript is in fact an implementation of a standardized scripting language called **ECMAScript**.

Internet Explorer (IE) at first did not support JavaScript, but instead had its own browser-based scripting language (VBScript). While IE now does support JavaScript, Microsoft sometimes refers to it as JScript, primarily for trademark rea- sons (Oracle currently owns the trademark for JavaScript).

It wasn't until the middle of the 2000s with the emergence of so-called AJAX sites that JavaScript became a much more important part of web development. **AJAX** is both an acronym as well as a general term. As an acronym it means Asynchronous JavaScript and XML, which was accurate for some time; but since XML is no longer always the data format for data transport in AJAX sites, the acronym meaning is becoming less and less accurate.

The most important way that this responsiveness is created is via asynchronous data requests via JavaScript and the **XMLHttpRequest** object. This addition to JavaScript was introduced by Microsoft as an ActiveX control (the IE version of plug-ins) in 1999, but it wasn't until sophisticated websites by Google (such as Gmail and Maps) and Flickr demonstrated what was possible using these techniques that the term AJAX became popular. The most important feature of AJAX sites is the asynchronous data requests.

As you can see in Figure 6.4, such interaction requires multiple requests to the server, which not only slows the user experience, it puts the server under extra load, especially if, as the case in Figure 6.4, each request is invoking a server-side script.

With ever-increasing access to processing power and bandwidth, sometimes it can be really hard to tell just how much impact these requests to the server have, but it's important to remember that more trips to the server do add up, and on a large scale this can result in performance issues.

But as can be seen in Figure 6.5, when these multiple requests are being made across the Internet to a busy server, then the time costs of the normal HTTP request- response loop will be more visually noticeable to the user.

AJAX provides web authors with a way to avoid the visual and temporal defi- ciencies of normal HTTP interactions. With AJAX web pages, it is possible to update sections of a page by making special requests of the server in the back- ground, creating the illusion of continuity. Figure 6.6 illustrates how the interaction shown in Figure 6.4 would differ in an AJAX-enhanced web page.

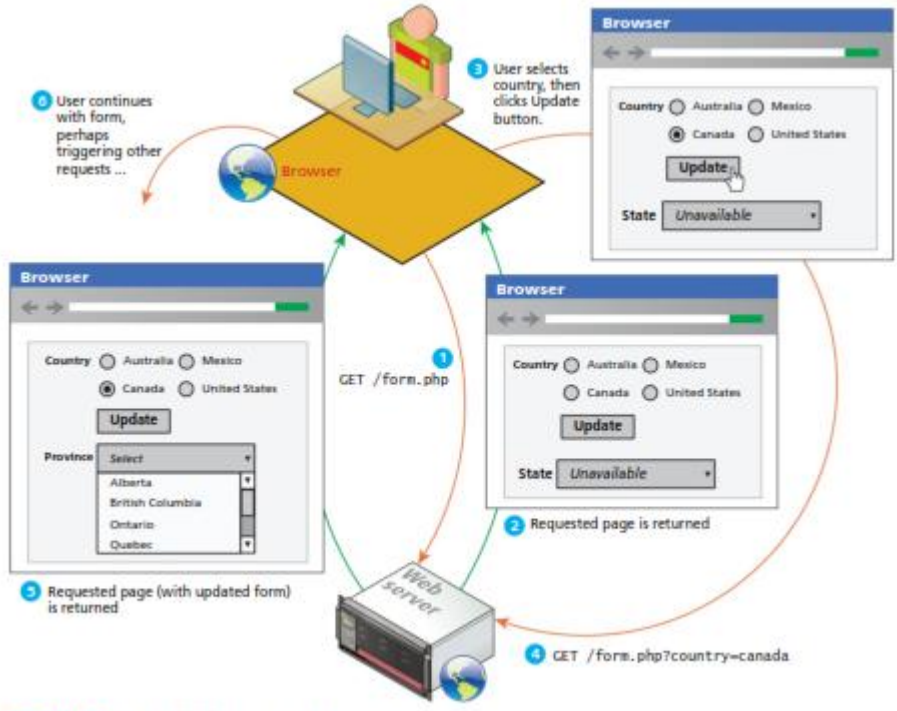

FIGURE 6.4 Normal HTTP request-response loop

Figure 6.7 illustrates some sample jQuery plug-ins, which are a way for developers to extend the functionality of jQuery. There are thousands of jQuery plug-ins available, which handle everything from additional user interface functionality to data handling.

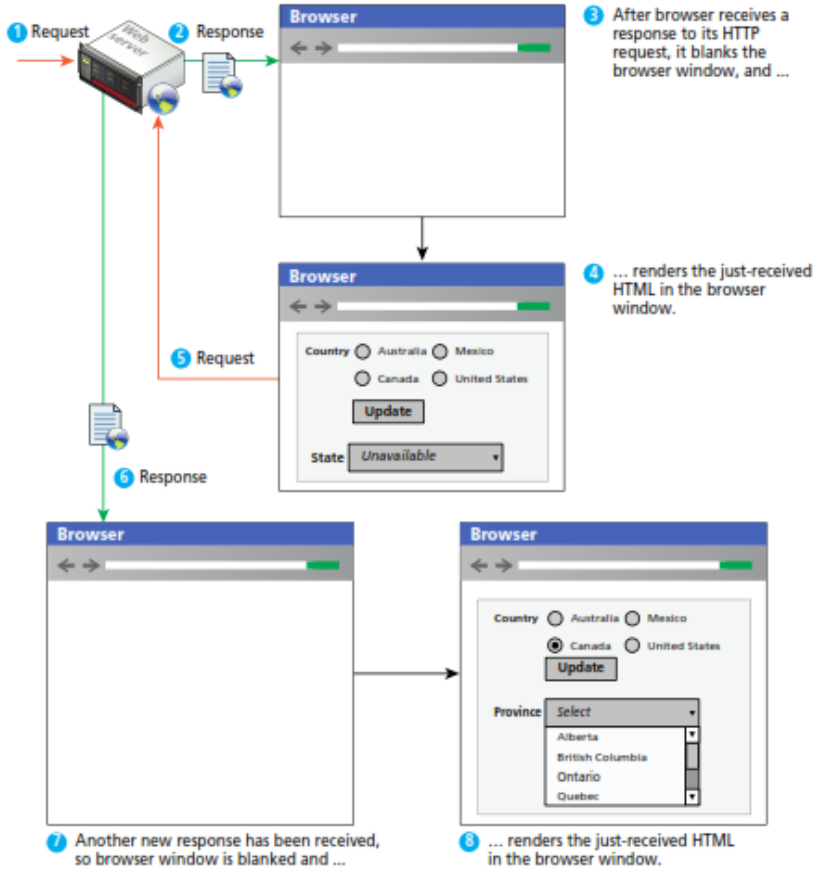

FIGURE 6.5 Normal HTTP request-response loop, take two

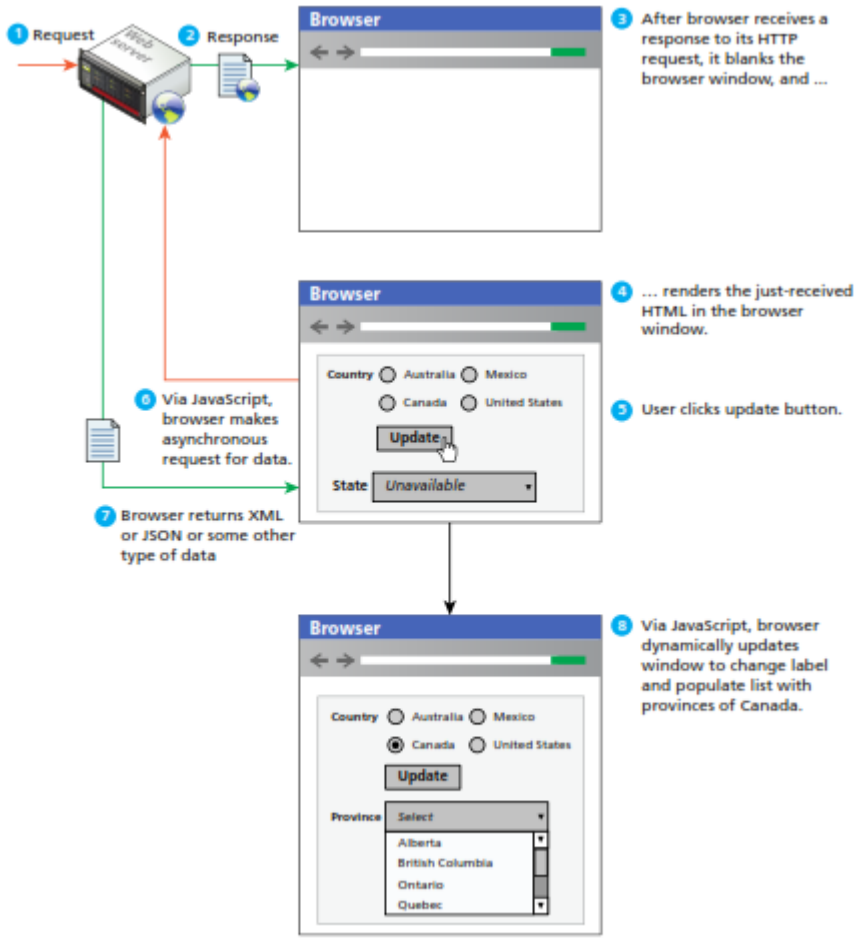

FIGURE 6.6 Asynchronous data requests

More recently, sophisticated MVC JavaScript frameworks such as AngularJS, Backbone, and Knockout have gained a lot of interest from developers wanting to move more data processing and handling from server-side scripts to HTML pages using a software engineering best practice, namely the separation of the model (data representation) from the view (presentation of data) design pattern.

3.2 Javascript Design principles

As mentioned earlier, JavaScript does have a bad reputation for being a difficult language to use. Although frameworks and developer tools can help, there is some truth to this reputation.

It should be said, however, that this reputation is based not so much on the language itself but in how developers have tended to use it. JavaScript has often been used through inline HTML hooks—that is, embedded into the HTML of a web page. Although this technique has been used for years, it has the distinct disadvantage- tage of blending HTML and JavaScript together, which decreases code readability, and increases the difficulty of web development.

3.2.1 Layers

When designing software to solve a problem, it is often helpful to abstract the solution a little bit to help build a cognitive model in your mind that you can then implement. Perhaps the most common way of articulating such a cognitive model is via the term **layer**. In object-oriented programming, a software **layer** is a way of conceptually grouping programming classes that have similar functionality and dependencies. Common software design layer names include:

- **Presentation layer.** Classes focused on the user interface.
- **Business layer.** Classes that model real-world entities, such as customers, products, and sales.
- **Data layer.** Classes that handle the interaction with the data sources.

We can say here simply that layers are a time-tested way to improve the quality and maintainability of software projects.

To help us conceptualize good design, we will consider JavaScript layers that exist both above and below pure HTML pages. These layers have different capa- bilities and responsibilities, but are always considered optional, except in some special circumstances like online games.

Although each layer can perform many tasks, it is helpful to visualize and understand the types of conceptual layers that are common. Figure 6.8 illustrates the idea of JavaScript layers.

Presentation Layer

This type of programming focuses on the display of information. JavaScript can alter the HTML of a page, which results in a change, visible to the user. These pre- sentation layer applications include common things like creating, hiding, and show- ing divs, using tabs to show multiple views, or having arrows to page through result sets. This layer is most closely related to the user experience and the most visible to the end user.

Validation Layer

JavaScript can be also used to validate logical aspects of the user's experience. This could include, for example, validating a form to make sure the email entered is valid before sending it along. It is often used in conjunction with the presentation layer

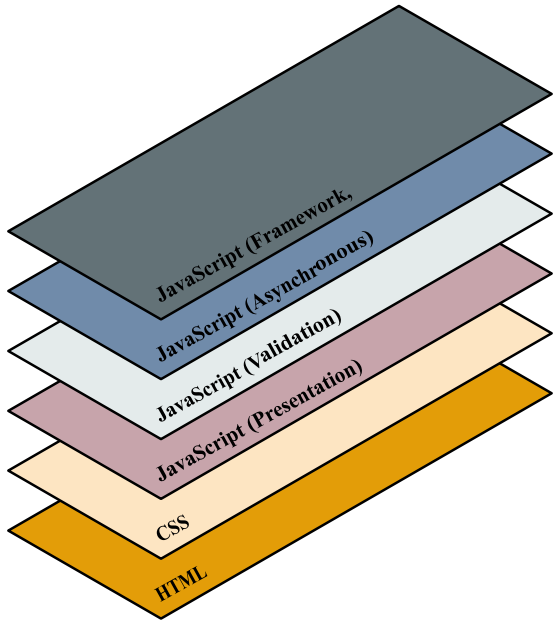

Figure 6.8 JavaScript layers

to create a coherent user experience, where a message to the presentation layer highlights bad fields. Both layers exist on the client machine, although the intention is to prevalidate forms before making transmissions back to the server.

Asynchronous Layers

Normally, JavaScript operates in a synchronous manner where a request sent to the server requires a response before the next lines of code can be executed. During the wait between request and response the browser sits in a loading state and only updates upon receiving the response. In contrast, an asynchronous layer can route requests to the server in the background. In this model, as certain events are triggered, the JavaScript sends the HTTP requests to the server, but while waiting for the response, the rest of the application functions normally, and the browser isn't in a loading state. When the response arrives JavaScript will (perhaps) update a portion of the page. Asynchronous layers are considered advanced versions of the presentation and validation layers above.

Typically developers work on a single file or application, weaving aspects of logic and presentation together. Although this is a common practice, separating the presentation and logic in your code is a powerful technique worth keeping in mind as you code. Having separate presentation and logic functions/classes will help you achieve more reusable code, which also happens to be easier to understand and maintain as illustrated in Figure 6.20.

3.2.2 Users without javascript

Too often website designers believe (erroneously) that users without JavaScript are somehow relics of a forgotten age, using decades-old computers in a bomb shelter somewhere philosophically opposed to updating their OS and browsers and there- fore not worth worrying about. Nothing could be more of a straw man argument. Users have a myriad of reasons for not using JavaScript, and that includes some of the most important clients, like search engines. A client may not have JavaScript because they are a web crawler, have a browser plug-in, are using a text browser, or are visually impaired.

.

- **Web crawler**. A web crawler is a client running on behalf of a search engine to download your site, so that it can eventually be featured in their search results. These automated software agents do not interpret JavaScript, since it is costly, and the crawler cannot see the enhanced look anyway.
- **Browser plug-in**. A browser plug-in is a piece of software that works within the browser, that might interfere with JavaScript. There are many uses of JavaScript that are not desirable to the end user. Many malicious sites use JavaScript to compromise a user's computer, and many ad networks deploy advertisements using JavaScript. This motivates some users to install plug-ins that stop JavaScript execution. An ad-blocking plug-in, for example, may filter JavaScript scripts that include the word *ad*, so a script named **advanced.js** would be blocked inadvertently.
- **Text-based client**. Some clients are using a text-based browser. Text-based browsers are widely deployed on web servers, which are often accessed using a command-line interface. A website administrator might want to see what
  an HTTP GET request to another server is returning for testing or support purposes. Such software includes lynx as shown in Figure 6.9.
- **Visually disabled client**. A visually disabled client will use special web browsing software to read the contents of a web page out loud to them. These specialized browsers do not interpret JavaScript, and some JavaScript on sites is not accessible to these users. Designing for these users requires some extra considerations, with lack of JavaScript being only one of them. Open-source browsers like WebIE would display the same site as shown in Figure 6.10.

the <Noscript> tag

Now that we know there are many sets of users that may have JavaScript disabled, we may want to make use of a simple mechanism to show them special HTML con- tent that will not be seen by those with JavaScript. That mechanism is the HTML tag <noscript>. Any text between the opening and closing tags will only be displayed to users without the ability to load JavaScript. It is often used to prompt users to enable JavaScript, but can also be used to show additional text to search engines.

FIGURE 6.9 Surfing the web with Lynx

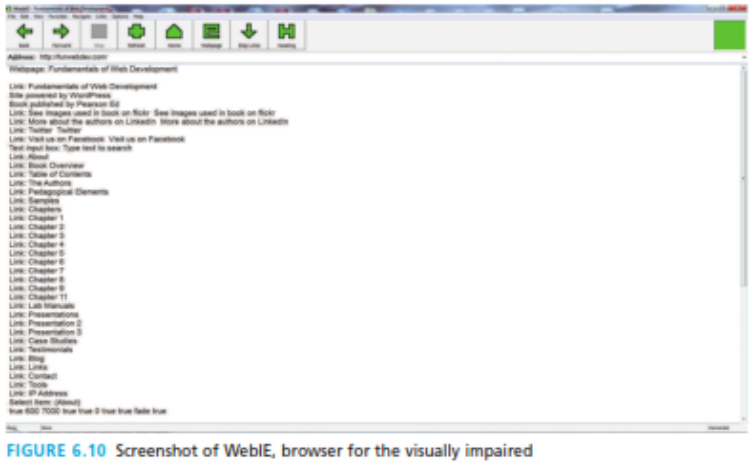

FIGURE 6.10 Screenshot of WebIE, browser for the visually impaired

Increasingly, websites that focus on JavaScript or Flash only risk missing out on an important element to help get them noticed: search engine optimization (SEO). Moreover, older or mobile browsers may not have a complete JavaScript implementation. Requiring JavaScript (or Flash) for the basic operation of your site will cause problems eventually and should be avoided. In this spirit, we should create websites with all the basic functionality enabled using regular HTML. For those (majority) of users with JavaScript enabled we can then enhance the basic layout using JavaScript to: embellish the look of certain ele- ments, animate certain user interactions, prevalidate forms, and generally replace static HTML elements with more visually and logically enhanced elements from JavaScript.

This approach of adding functional replacements for those without JavaScript is also referred to as **fail-safe design**, which is a phrase with a meaning beyond web development. It means that when a plan (such as displaying a fancy JavaScript pop- up calendar widget) fails (because for instance JavaScript is not enabled), then the system's design will still work .

3.2.3 Graceful Degradation and progressive enhancement

The principle of fail-safe design can still apply even to browsers that have enabled JavaScript. Over the years, browser support for different JavaScript objects has varied. Something that works in the current version of Chrome might not work in IE version 8; something that works in a desktop browser might not work in a mobile browser. In such cases, what strategy should we take as web application developers?

The principle of **graceful degradation** is one possible strategy. With this strategy you develop your site for the abilities of current browsers. For those users who are not using current browsers, you might provide an alternate site or pages for those using older browsers that lack the JavaScript (or CSS or HTML5) used on the main site.

The alternate strategy is **progressive enhancement**, which takes the opposite approach to the problem. In this case, the developer creates the site using CSS, JavaScript, and HTML features that are supported by all browsers of a certain age or newer. (Eventually, one does have to stop supporting ancient browsers; many developers have, for instance, stopped supporting IE 6.) To that baseline site, the developers can now "progressively" (i.e., for each browser) "enhance" (i.e., add functionality) to their site based on the capabilities of the users' browsers.

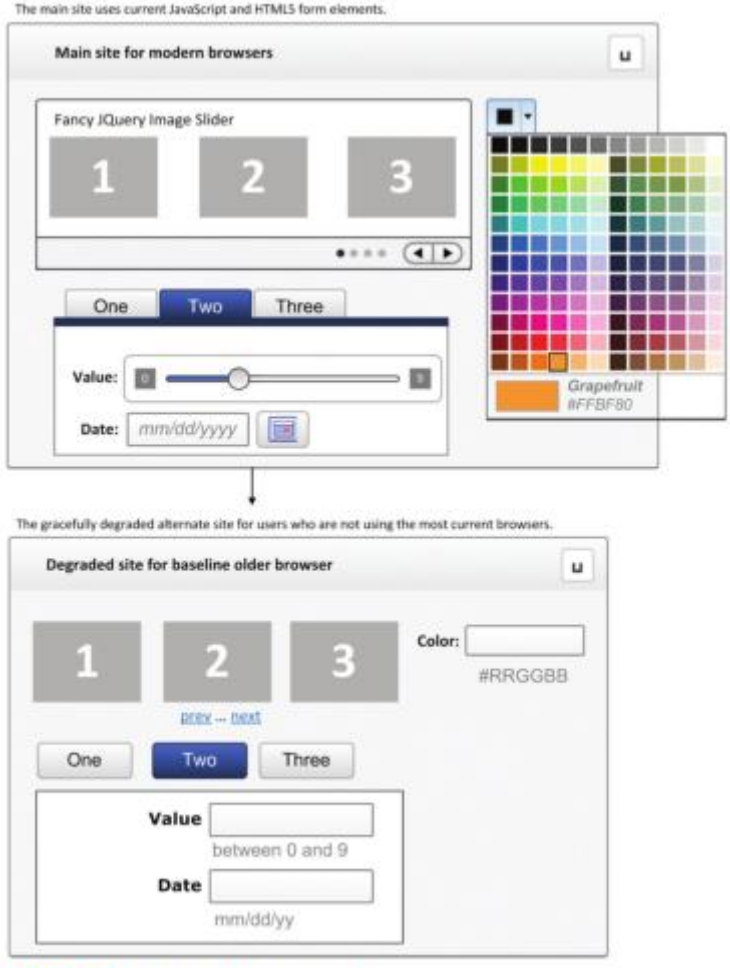

FIGURE 6.11 Example of graceful degradation

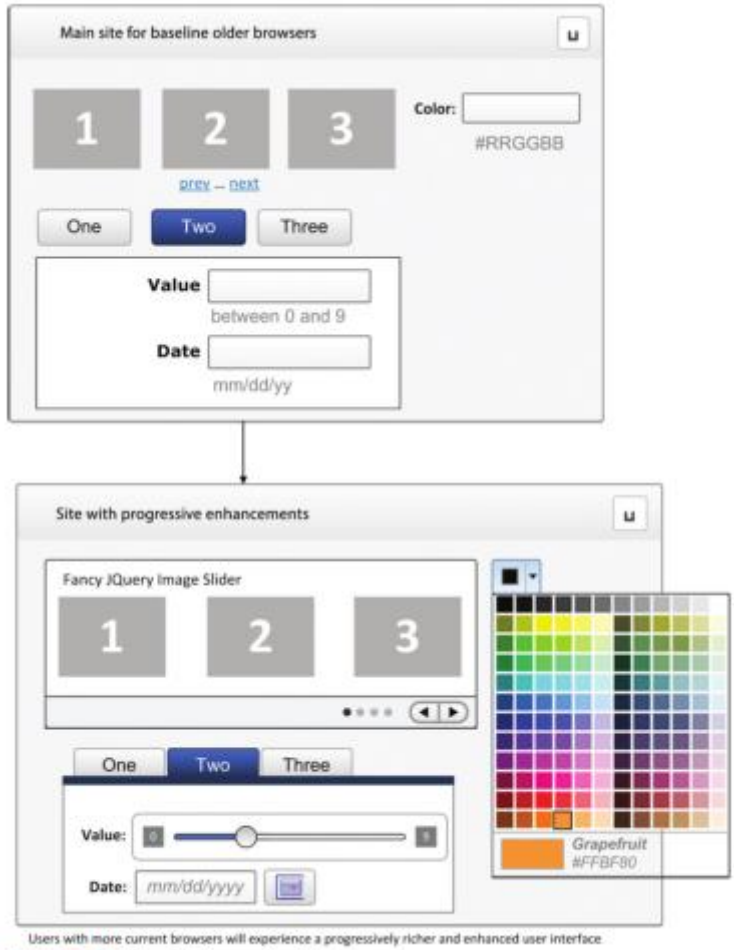

FIGURE 6.12 Site with Progressive Enhancements

3.3 Where Does javascript Go?

JavaScript can be linked to an HTML page in a number of ways. Just as CSS styles can be **inline**, **embedded**, or **external**, JavaScript can be included in a number of ways. Just as with CSS these can be combined, but external is the preferred method for cleanliness and ease of maintenance.

Running JavaScript scripts in your browser requires downloading the JavaScript code to the browser and then running it. Pages with lots of scripts could potentially run slowly, resulting in a degraded experience while users wait for the page to load. Different browsers manage the downloading and loading of scripts in different ways, which are important things to realize when you decide how to link your scripts.

Inline JavaScript

Inline JavaScript refers to the practice of including JavaScript code directly within certain HTML attributes Inline JavaScript is a real maintenance nightmare

```
<a href="JavaScript:OpenWindow();"more info</a>
<input type="button" onclick="alert('Are you sure?');" />
```

LISTING 6.1 Inline JavaScript example

Embedded JavaScript

Embedded JavaScript refers to the practice of placing JavaScript

code within a <script> element

```
<script type="text/javascript">
/* A JavaScript Comment */
alert ("Hello World!");
</script>
```

LISTING 6.2 Embedded JavaScript example

External JavaScript

JavaScript supports this separation by allowing links to an external file that contains the JavaScript.

By convention, JavaScript external files have the extension .js.

```
<head>
  <script type="text/JavaScript" src="greeting.js">
  </script>
</head>
```

LISTING 6.3 External JavaScript example

3.4 syntax

Since it's a lightweight scripting language, JavaScript has some features (such as dynamic typing) that are especially helpful to the novice programmer. However, a novice programmer faces challenges when he or she tries to use JavaScript in the same way as a full object-oriented language such as Java, as JavaScript's object features (such as prototypes and inline functions) are quite unlike those of more familiar languages.

We will briefly cover the fundamental syntax for the most common program- ming constructs including **variables**, **assignment**, **conditionals**, **loops**, and **arrays** before moving on to advanced topics such as events and classes.

JavaScript's reputation for being quirky not only stems from its strange way of implementing object-oriented principles, but also from some odd syntactic *gotchas* that every JavaScript developer will eventually encounter, some of which include:

Everything is type sensitive, including function, class, and variable names.

- The scope of variables in blocks is not supported. This means variables declared inside a loop may be accessible outside of the loop, counter to what one would expect.

- There is a === operator, which tests not only for equality but type equivalence.
- Null and undefined are two distinctly different states for a variable. Semicolons are not required, but are permitted (and encouraged).
- There is no integer type, only number, which means floating-point rounding errors are prevalent even with values intended to be integers.

```
var x;      <-- a variable x is defined
var y = 0;  <-- y is defined and initialized to 0
y = 4;      <-- y is assigned the value of 4
```

FIGURE 6.13 Variable declaration and assignment

3.4.1 variables

Variables in JavaScript are **dynamically typed**, meaning a variable can be an integer, and then later a string, then later an object, if so desired. This simplifies variable declarations, so that we do not require the familiar type fields like *int*, *char*, and *String*. Instead, to declare a variable x, we use the var keyword, the name, and a semicolon as shown in Figure 6.13. If we specify no value, then (being typeless) the default value is undefined.

Assignment can happen at declaration-time by appending the value to the declaration, or at run time with a simple right-to-left assignment as illustrated in Figure 6.13. This syntax should be familiar to those who have programmed in languages like C and Java.

In addition, the **conditional assignment** operator, shown in Figure 6.14, can also be used to assign based on condition, although its use is sometimes discouraged.

3.4.2 Comparison Operators

The core of any programming language is the ability to distill things down to Boolean statements where something is either true or false. JavaScript is no excep- tion and comes equipped with a number of operators to compare two values, listed in Table 6.1.

Operator	Description	Matches (x=9)
==	Equals	(x==9) is true (x=="9") is true
===	Exactly equals, including type	(x==="9") is false (x===9) is true
< , >	Less than, greater than	(x<5) is false
<= , >=	Less than or equal, greater than or equal	(x<=9) is true
!=	Not equal	(4!=x) is true
!==	Not equal in either value or type	(x!=="9") is true (x!==9) is false

TABLE 6.1 Comparison Operators

3.4.3 Logical Operators

Comparison operators are useful, but without being able to combine several together, their usefulness would be severely limited. Therefore, like most languages JavaScript includes Boolean operators, which allow us to build complicated expres- sions. The Boolean operators and, or, and not and their truth tables are listed in Table 6.2. Syntactically they are represented with && (and), || (or), and ! (not).

3.4.4 Conditionals

JavaScript's syntax is almost identical to that of PHP, Java, or C when it comes to conditional structures such as `if` and `if` else statements. In this syntax the condi- tion to test is contained within `( )` brackets with the body contained in `{ }` blocks.

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

AND Truth Table

A	B	A || B
T	T	T
T	F	T
F	T	T
F	F	F

OR Truth Table

A	! A
T	F
F	T

NOT Truth Table

TABLE 6.2 AND, OR, and NOT Truth Tables

```
var hourOfDay;   // var to hold hour of day, set it later...
var greeting;    // var to hold the greeting message.
if (hourOfDay > 4 && hourOfDay < 12){
   // if statement with condition
   greeting =  "Good Morning";
}
else if (hourOfDay >= 12 && hourOfDay < 20){
   // optional else if
   greeting =  "Good Afternoon";
}
else{ // optional else branch
   greeting = "Good Evening";
}
```

LISTING 6.4 Conditional statement setting a variable based on the hour of the day

3.4.5 Loops

Like conditionals, loops use the ( ) and { } blocks to define the condition and the body of the loop.

While Loops

The most basic loop is the while loop, which loops until the condition is not met. Loops normally initialize a **loop control variable** before the loop, use it in the condi- tion, and modify it within the loop. One must be sure that the variables that make up the condition are updated inside the loop (or elsewhere) to avoid an infinite loop!

```
var i=0;
while(i < 10){
 //do something with i
 i++;
}
```

For Loops

A **for loop** combines the common components of a loop: initialization, condition, and post-loop operation into one statement. This statement begins with the for keyword and has the components placed between ( ) brackets, semicolon (;) sepa- rated as shown in Figure 6.15.

```
for (var i = 0; i < 10; i++){
  //do something with i
}
```

FIGURE 6.15 For loop

3.4.6 Functions

Functions are the building block for modular code in JavaScript, and are even used to build **pseudo-classes**, which you will learn about later. They are defined by using the reserved word `function` and then the function name and (optional) parameters. Since JavaScript is dynamically typed, functions do not require a return type, nor do the parameters require type. Therefore a function to raise x to the yth power might be defined as:

```
function power(x,y){
  var pow=1;
  for (var i=0;i<y;i++){
     pow = pow*x;
  }
  return pow;
}
```

And called as

```
power(2,10);
```

With new programmers there is often confusion between defining a function and calling the function. Remember that when actually using the keyword `function`, we are defining what the function does. Later, we can use or call that function by using its given name *without* the function keyword.

alert

The `alert()` function makes the browser show a pop-up to the user, with whatever is passed being the message displayed. The following JavaScript code displays a simple hello world message in a pop-up:

```
alert( "Good Morning" );
```

The pop-up may appear different to each user depending on their browser con- figuration. What is universal is that the pop-up obscures the underlying web page, and no actions can be done until the pop-up is dismissed.

Alerts are not used in production code, but are a useful tool for debugging and illustration purposes. Alerts are used throughout the chapter to provide example output, and in practice are often used for debugging or as placeholders for the even- tual code, which might log to a file, transmit a message, or update an interface.

3.4.7 errors Using try and Catch

When the browser's JavaScript engine encounters an error, it will *throw* an **exception**. These exceptions interrupt the regular, sequential execution of the pro- gram and can stop the JavaScript engine altogether. However, you can optionally catch these errors preventing disruption of the program using the **try–catch block** as shown below

```
try
{
nonexistantfunction("hello");
}
catch(err)
{
alert("An exception was caught:" + err);
}
```

throwing Your Own exceptions

Although try-catch can be used exclusively to catch built-in JavaScript errors, it can also be used by your programs, to throw your own messages. The throw key- word stops normal sequential execution, just like the built-in exceptions as shown in Listing 6.6.
The general consensus in software development is that try-catch and throw statements should be used for *abnormal* or *exceptional* cases in your program. They

should not be used as a normal way of controlling flow, although no formal mech- anism exists to enforce that idea. We will generally avoid try-catch statements in our code unless illustrative of some particular point. Listing 6.6 demonstrates the throwing of a user-defined exception as a string. In reality any object can be thrown, although in practice a string usually suffices.

```
try {
    var x = -1;
    if (x<0)
    throw "smallerthan0Error";
    }
    catch(err){
        alert (err + "was thrown");
    }
```

Listing 6.6 Throwing a user-defined exception

It should be noted that throwing an exception disrupts the sequential execution of a program. That is, when the exception is thrown all subsequent code is not executed until the catch statement is reached. This reinforces why try-catch is for exceptional cases.

3.5 JavaScript Objects

JavaScript is not a full-fledged object-oriented programming language. It does not have classes per se, and it does not support many of the patterns you'd expect from an object-oriented language like inheritance and polymorphism in a straightforward way.

The language does, however, support objects. User-defined objects are declared in a slightly odd way to developers familiar with languages like C++ or Java, so the syntax to build pseudo-classes can be challenging. Nonetheless the advantages of encapsulating data and methods into objects outweigh the syntactic hurdle you will have to overcome.

Objects can have **constructors**, **properties**, and **methods** associated with them, and are used very much like objects in other object-oriented languages. There are objects that are included in the JavaScript language; you can also define your own kind of objects.

3.5.1 Constructors

Normally to create a new object we use the new and
inside, comma delimited as follows:

```
var someObject = new ObjectName(parameter 1,param 2,..., parameter n);
```

For some classes, shortcut constructors are defined, which can be confusing if we are not aware of them. For example, a String object can be defined with the shortcut

```
var greeting = "Good Morning";
```

Instead of the formal definition

```
var greeting = new String("Good Morning");
```

Arrays are another class with a shortcut constructor,

3.5.2 properties

Each object might have properties that can be accessed, depending on its definition. When a property exists, it can be accessed using **dot notation** where a dot between the instance name and the property references that property.

```
alert(someObject.property);//show someObject.property to the user
```

Methods

Objects can also have methods, which are functions associated with an instance of an object. These methods are called using the same dot notation as for properties, but instead of accessing a variable, we are calling a method.

```
someObject.doSomething();
```

Methods may produce different output depending on the object they are associ- ated with because they can utilize the internal properties of the object.

3.5.3 Objects included in javascript

A number of useful objects are included with JavaScript. These include Array, Boolean, Date, Math, String, and others. In addition to these, JavaScript can also access Document Object Model (DOM) objects that correspond to the content of a page's HTML. These DOM objects let JavaScript code access and modify HTML and CSS properties of a page dynamically.

arrays

Arrays are one of the most used data structures, and they have been included in JavaScript as well. In practice, this class is defined to behave more like a linked list in that it can be resized dynamically, but the implementation is browser specific, meaning the efficiency of insert and delete operations is unknown.

Arrays will be the first objects we will examine. Objects can be created using the new syntax and calling the object constructor. The following code creates a new, empty array named greetings:

```
var greetings = new Array();
```

To initialize the array with values, the variable declaration would look like the following:

```
var greetings = new Array("Good Morning", "Good Afternoon");
```

or, using the square bracket notation:

```
var greetings = ["Good Morning", "Good Afternoon"];
```

While you should be careful to employ consistency in your own array declara- tions, it's important to familiarize yourself with notation that may be used by others. Teams should agree on some standards in this area.

Accessing and Traversing an Array

To access an element in the array you use the familiar square bracket notation from
Java and C-style languages, with the index you wish to access inside the brackets.

```
alert( greetings[0] );
```

One of the most common actions on an array is to traverse through the items sequentially. The following for loop quickly loops through an array, accessing the ith element each time using the Array object's length property to determine the maximum valid index. It will alert "Good Morning" and "Good Afternoon" to the user.

```
for (var i = 0; i < greetings.length; i++)
{
                    alert(greetings[i]);
}
```

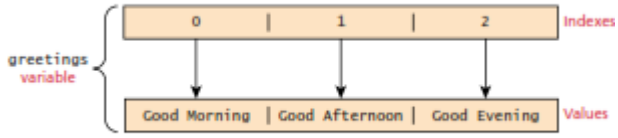

FIGURE 6.16 JavaScript array with indexes and values illustrated

Modifying an Array

To add an item to an existing array, you can use the push method.

```
greetings.push("Good Evening");
```

Figure 6.16 illustrates an array with indexes and the corresponding values.

The pop method can be used to remove an item from the back of an array. Additional methods that modify arrays include concat(), slice(), join(), reverse(), shift(), and sort(). A full accounting of all these methods is beyond the scope of a single chapter, but as you begin to use arrays you should explore them.

Math

The **Math class** allows one to access common mathematic functions and common values quickly in one place. This static class contains methods such as max(), min(), pow(), sqrt(), and exp(), and trigonometric functions such as sin(), cos(), and arctan(). In addition, many mathematical constants are defined such as PI, E (Euler's number), SQRT2, and some others as shown in Listing 6.7.

```
Math.PI              // 3.141592657
Math.sqrt(4);        // square root of 4 is 2.
Math.random();       // random number between 0 and 1
```

Listing 6.7 Some constants and functions in the Math object

string

The **String class** has already been used without us even knowing it. That is because it is core to communicating with the user. Since it is so common, shortcuts have been defined for creating and concatenating strings. While one can use the new syntax to create a String object, it can also be defined using quotes as follows:

```
var greet = new String("Good");     // long form constructor
var greet = "Good";                 // shortcut constructor
```

A common need is to get the length of a string. This is achieved through the length property (just as in arrays).

```
alert (greet.length); // will display "4"
```

Another common way to use strings is to concatenate them together. Since this is so common, the + operator has been overridden to allow for concatenation in place.

```
var str = greet.concat("Morning");     // Long form concatenation
var str = greet + "Morning";           // + operator concatenation
```

Many other useful methods exist within the String class, such as accessing a single character using charAt(), or searching for one using indexOf(). Strings allow splitting a string into an array, searching and matching with split(), search(), and match() methods.

Date

While not critical to JavaScript, the Date class is yet another helpful included object you should be aware of. It allows you to quickly calculate the current date or create date objects for particular dates. To display today's date as a string, we would sim- ply create a new object and use the toString() method.

```
var d = new Date();

// This outputs Today is Mon Nov 12 2012 15:40:19 GMT-0700
alert ("Today is "+ d.toString());
```

3.5.4 Window Object

The window object in JavaScript corresponds to the browser itself. Through it, you can access the current page's URL, the browser's

history, and what's being displayed in the status bar, as well as opening new browser windows. In fact, the `alert()` function mentioned earlier is actually a method of the window object.

3.6 The Document Object Model (DOM)

JavaScript is almost always used to interact with the HTML document in which it is contained. As such, there needs to be some way of programmatically accessing the elements and attributes within the HTML. This is accomplished through a programming interface (API) called the **Document Object Model (DOM)**.

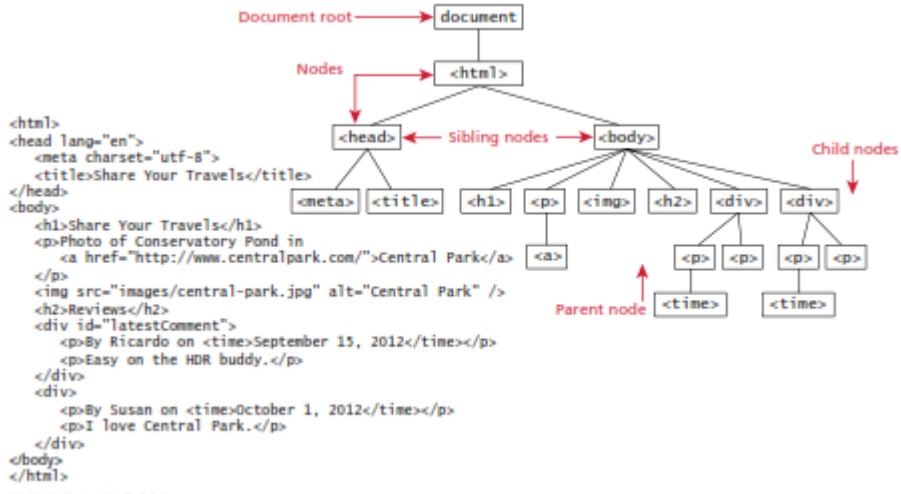

FIGURE 6.17 DOM tree

According to the W3C, the DOM is a:

We already know all about the DOM, but by another name. The tree structure from Chapter 2 (shown again in Figure 6.17) is formally called the **DOM Tree** with the root, or topmost object called the **Document Root**. You already know how to specify the style of documents using CSS; with JavaScript and the DOM, you now can do so dynamically as well at run time, in response to user events.

3.6.1 Nodes

In the DOM, each element within the HTML document is called a **node**. If the DOM is a tree, then each node is an individual branch. There are element nodes, text nodes, and attribute nodes, as shown in Figure 6.18.

All nodes in the DOM share a common set of properties and methods. Thus, most of the tasks that we typically perform in JavaScript involve finding a node, and then accessing or modifying it via those properties and methods. The most impor- tant of these are shown in Table 6.3.

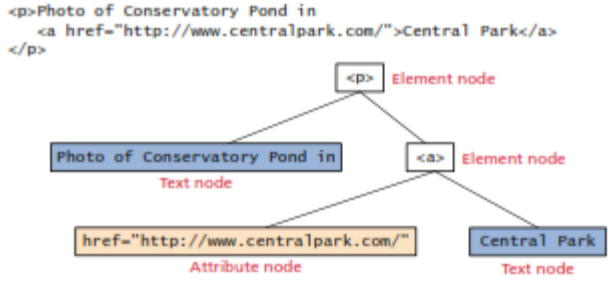

FIGURE 6.18 DOM nodes

Property	Description
attributes	Collection of node attributes
childNodes	A NodeList of child nodes for this node
firstChild	First child node of this node
lastChild	Last child of this node
nextSibling	Next sibling node for this node
nodeName	Name of the node
nodeType	Type of the node
nodeValue	Value of the node
parentNode	Parent node for this node
previousSibling	Previous sibling node for this node.

TABLE 6.3 Some Essential Node Object Properties

3.6.2 Document Object

The **DOM document object** is the root JavaScript object representing the entire HTML document. It contains some properties and methods that we will use exten- sively in our development and is globally accessible as document. The attributes of this object include some information about the page including doctype and inputEncoding. Accessing the properties is done through the dot nota- tion as illustrated on the next page.

Method	Description
`createAttribute()`	Creates an attribute node
`createElement()`	Creates an element node
`createTextNode()`	Creates a text node
`getElementById(id)`	Returns the element node whose `id` attribute matches the passed `id` parameter
`getElementsByTagName(name)`	Returns a NodeList of elements whose tag name matches the passed name parameter

TABLE 6.4 Some Essential Document Object Methods

```
// specify the doctype, for example html
var a = document.doctype.name;
// specify the page encoding, for example ISO-8859-1
var b = document.inputEncoding;
```

In addition to these moderately useful properties, there are some essential meth- ods (see Table 6.4) you will use all the time. They include getElementByTagName() and the indispensable getElementById(). While the former method returns an array of DOM nodes (called a NodeList) matching the tag, the latter returns a single DOM element (covered below), that matches the id passed as a parameter as illus- trated in Figure 6.19.

FIGURE 6.19 Relationship between HTML tags and getElementByID() and getElementsByTagName()

Selectors are generally poorly supported in pure JavaScript across the multitude of browsers and platforms available. The method getElementById() is universally implemented and thus used extensively. The newer querySelector() and querySelectorAll() methods allow us to query for DOM elements much the same way we specify CSS styles, but are only implemented in the newest browsers. we will rely on getElementByID().

3.6.3 element Node Object

The type of object returned by the method document.getElementById() described in the previous section is an **element node** object. This represents an HTML element in the hierarchy, contained between the opening <> and closing </> tags for this element. As you may already have figured out, an element can itself contain more elements.

Since IDs must be unique in an HTML document, getElementByID() returns a single node, rather than a set of results which is the case with other selector functions. The returned Element Node object has the node properties shown in Table 6.3. It also has a variety of additional properties, the most important of which are shown in Table 6.5.

While these properties are available for all HTML elements, there are some HTML elements that have additional properties that can be accessed. Table 6.6 lists some common additional properties and the HTML tags that have these properties.

3.6.4 Modifying a DOM element

In many introductory JavaScript textbooks the document.write() method is used to create output to the HTML page from JavaScript. While this is certainly valid, it

Property	Description
className	The current value for the `class` attribute of this HTML element.
id	The current value for the `id` of this element.
innerHTML	Represents all the things inside of the tags. This can be read or written to and is the primary way in which we update particular elements using JavaScript.
style	The `style` attribute of an element. We can read and modify this property.
tagName	The tag name for the element.

TABLE 6.5 Some Essential Element Node Properties

I n be read or written to and is

Property	Description	Tags
href	The href attribute used in a tag to specify a URL to link to.	`a`
name	The `name` property is a bookmark to identify this tag. Unlike `id`, which is available to all tags, `name` is limited to certain form-related tags.	`a, input, textarea, form`
src	Links to an external URL that should be loaded into the page (as opposed to `href`, which is a link to follow when clicked)	`img, input, iframe, script`
value	The value is related to the `value` attribute of input tags. Often the value of an input field is user defined, and we use `value` to get that user input.	`input, textarea, submit`

TABLE 6.6 Some Specific HTML DOM Element Properties for Certain Tag Types

always creates JavaScript at the bottom of the existing HTML page, and in practice is good for little more than debugging. The modern JavaScript programmer will want to write to the HTML page, but in a particular location, not always at the bottom.

Using the DOM document and HTML DOM element objects, we can do exactly that using the innerHTML property as shown in Listing 6.8 (using the HTML shown in Figure 6.19).

```
var latest = document.getElementById("latestComment");
var oldMessage = latest.innerHTML;
latest.innerHTML = oldMessage + "<p>Updated this div with JS</p>";
```

LISTING 6.8 Changing the HTML using innerHTML

Now the HTML of our document has been modified to reflect that change.

```
<div id="latestComment">
   <p>By Ricardo on <time>September 15, 2012</time></p>
   <p>Easy on the HDR buddy.</p>
   <p>Updated this div with JS</p>
</div>
```

A More verbose technique

Although the innerHTML technique works well (and is very fast), there is a more verbose technique available to us that builds output using the DOM. This more explicit technique has the advantage of ensuring that only valid markup is created, while the innerHTML could output badly formed HTML. DOM functions create- TextNode(), removeChild(), and appendChild() allow us to modify an element in a more rigorous way as shown in Listing 6.9.

```
var latest = document.getElementById("latestComment");
var oldMessage = latest.innerHTML;
var newMessage = oldMessage + "<p>Updated this div with JS</p>";
latest.removeChild(latest.firstChild);
latest.appendChild(document.createTextNode(newMessage));
```

LISTING 6.9 Changing the HTML using createTextNode( ) and appendChild( )

Changing an element's style

We can also modify the style associated with a particular block. We can add or remove any style using the style or className property of the Element node, which is something that you might want to do to dynamically change the appear- ance of an element. Its usage is shown below to change a node's background color and add a three-pixel border.

```
var commentTag = document.getElementById("specificTag");
commentTag.style.backgroundColour = "#FFFF00";
commentTag.style.borderWidth="3px";
```

Armed with knowledge of CSS attributes you can easily change any style attribute. Note that the style property is itself an object, specifically a CSSStyleDeclaration type, which includes all the CSS attributes as properties and computes the current style from inline, external, and embedded styles. Although you can directly access CSS style elements we suggest you use classes whenever possible.

```
<input type='password' name='pw' id='pw' />

We would use the following JavaScript code:

var pass = document.getelementById("pw");
alert (pass.value);
```

The className property is normally a better choice, because it allows the styles to be created outside the code, and thus be better accessible to designers. Using this model we would change the background color by having two styles defined, and changing them in JavaScript code.

```
var commentTag = document.getElementById("specificTag");
commentTag.className = "someClassName";
```

HTML5 introduces the classList element, which allows you to add, remove, or toggle a CSS class on an element. You could add a class with

```
label.classList.addClass("someClassName");
```

3.6.5 Additional properties

In addition to the global properties present in all tags, there are additional methods available when dealing with certain tags. Table 6.6 lists a few common ones. To get the password out of the following input field and alert the user

It should be obvious how getting the src or href properties out of appropriate tags could also be done. We leave it as an exercise to the reader.

3.7 Javascript events

At the core of all JavaScript programming is the concept of an event. A JavaScript event is an action that can be detected by JavaScript. Many of them are initiated by user actions but some are generated by the browser itself. We say then that an event is triggered and then it can be caught by JavaScript functions, which then do something in response.In the original JavaScript world, events could be specified right in the HTML markup with hooks to the JavaScript code (and still can).

This mechanism was popular throughout the 1990s and 2000s because it worked. As more powerful frameworks were developed, and website design and best practices were refined, this original mechanism was supplanted by the listener approach. A visual comparison of the old and new technique is shown in Figure 6.20.

Note how the old method weaves the JavaScript right inside the HTML, while the listener technique has removed JavaScript from the markup, resulting in cleaner, easier to maintain HTML code.

3.7.1 Inline event handler approach

JavaScript events allow the programmer to react to user interactions. In early web development, it made sense to weave code and HTML together and to this day, inline JavaScript calls are intuitive. For example, if you wanted an alert to pop-up when clicking a you might program:

```
<div id="example1" onclick="alert('hello')">Click for pop-up</div>
```

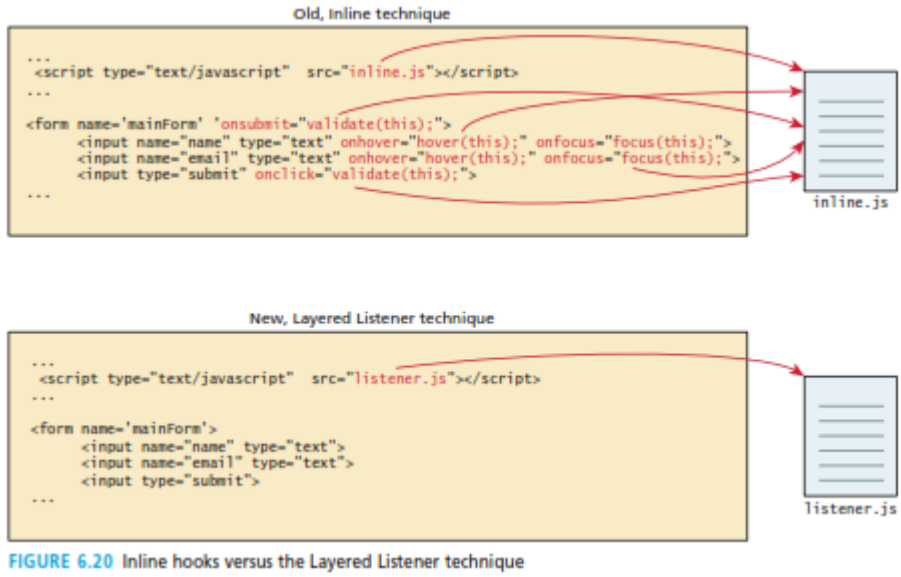

FIGURE 6.20 Inline hooks versus the Layered Listener technique

3.7.2 Listener approach

Section 6.2.1 argued that the design principle of layers is a proven way of increasing maintainability and simplifying markup. The problem with the inline handler approach is that it does not make use of layers; that is, it does not separate content from behavior.

For this reason, this book will advocate and use an approach that separates all JavaScript code from the HTML markup. Although the book and its labs may occa- sionally illustrate a quick concept with the old-style inline handler approach, the authors prefer to replace the inline approach using one of the two approaches shown in Listing 6.10 and Listing 6.11.

```
var greetingBox = document.getElementById('example1');
greetingBox.onclick = alert('Good Morning');
```

LISTING 6.10 The "old" style of registering a listener.

The approach shown in Listing 6.10 is widely supported by all browsers. The first line in the listing creates a temporary variable for the HTML element that will trigger the event. The next line attaches the element's `onclick` event to the event handler, which invokes the JavaScript `alert()` method (and thus annoys the user with a pop-up hello message).

The main advantage of this approach is that this code can be written anywhere, including an external file that helps *uncouple* the HTML from the JavaScript. However, the one limitation with this approach (and the inline approach) is that only one handler can respond to any given element event.

```
var greetingBox = document.getElementById('example1');
greetingBox.addEventListener('click', alert('Good Morning'));
greetingBox.addEventListener('mouseOut', alert('Goodbye'));

// IE 8
greetingBox.attachEvent('click', alert('Good Morning'));
```

LISTING 6.11 The "new" DOM2 approach to registering listeners.

The use of addEventListener() shown in Listing 6.11 was introduced in DOM Version 2, and as such is unfortunately not supported by IE 8 or earlier. This approach has all the other advantages of the approach shown in Listing 6.10, and has the additional advantage that multiple handlers can be assigned to a single object's event.

The examples in Listing 6.10 and Listing 6.11 simply used the built-in JavaScript alert() function. What if we wanted to do something more elaborate when an event is triggered? In such a case, the behavior would have to be encapsulated within a function, as shown in Listing 6.12.

```
function displayTheDate() {
   var d = new Date();
   alert ("You clicked this on "+ d.toString());
}
var element = document.getElementById('example1');
element.onclick = displayTheDate;

// or using the other approach
element.addEventListener('click',displayTheDate);
```

LISTING 6.12 Listening to an event with a function

An alternative to that shown in Listing 6.12 is to use an anonymous function (that is, one without a name), as shown in Listing 6.13. This approach is especially common when the event handling function will only ever be used as a listener.

```
var element = document.getElementById('example1');
element.onclick = function() {
   var d = new Date();
   alert ("You clicked this on " + d.toString());
};
```

LISTING 6.13 Listening to an event with an anonymous function

3.7.3 event Object

No matter which type of event we encounter, they are all **DOM event objects** and the event handlers associated with them can access and manipulate them. Typically we see the events passed to the function handler as a parameter named *e*.

```
function someHandler(e) {
   // e is the event that triggered this handler.
}
```

These objects have many properties and methods. Many of these properties are not used, but several key properties and methods of the event object are worth knowing.

Bubbles. The bubbles property is a Boolean value. If an event's bubbles property is set to true then there must be an event handler in place to handle the event or it will bubble up to its parent and trigger an event handler there. If the parent has no handler it continues to bubble up until it hits the document root, and then it goes away, unhandled.

Cancelable. The Cancelable property is also a Boolean value that indicates whether or not the event can be cancelled. If an event is cancelable, then the default action associated with it can be canceled. A common example is a user clicking on a link. The default action is to follow the link and load the new page.

preventDefault. A cancelable default action for an event can be stopped using the preventDefault()method as shown in Listing 6.14. This is a common practice when you want to send data asynchronously when a form is submitted, for example, since the default event of a form submit click is to post to a new URL (which causes the browser to refresh the entire page).

```
function submitButtonClicked(e) {
  if (e.cancelable){
    e. preventDefault();
  }
}
```

LISTING 6.14 A sample event handler function that prevents the default event

3.7.4 Event types

Perhaps the most obvious event is the click event, but JavaScript and the DOM sup- port several others. In actuality there are several classes of event, with several types of event within each class specified by the W3C. The classes are mouse events, keyboard events, form events, and frame events.

Mouse events

Mouse events are defined to capture a range of interactions driven by the mouse. These can be further categorized as mouse click and mouse move events. Table 6.7 lists the possible events one can listen for from the mouse.

Interestingly, many mouse events can be sent at a time. The user could be mov- ing the mouse off one and onto another in the same moment, triggering onmouseon and onmouseout events as well as the onmousemove event. The Cancelable and Bubbles properties can be used to handle these complexities.

Keyboard events

Keyboard events are often overlooked by novice web developers, but are important tools for power users. Table 6.8 lists the possible keyboard events.

Event	Description
onclick	The mouse was clicked on an element
ondblclick	The mouse was double clicked on an element
onmousedown	The mouse was pressed down over an element
onmouseup	The mouse was released over an element
onmouseover	The mouse was moved (not clicked) over an element
onmouseout	The mouse was moved off of an element
onmousemove	The mouse was moved while over an element

TABLE 6.7 Mouse Events in JavaScript

These events are most useful within input fields. We could for example validate an email address, or send an asynchronous request for a dropdown list of sugges- tions with each key press.

<input type="text" id="keyExample">

The input box above, for example, could be listened to and each key pressed echoed back to the user as an alert as shown in Listing 6.15.

```
document.getElementById("keyExample").onkeydown = function
myFunction(e){
  var keyPressed=e.keyCode;      //get the raw key code
  var character=String.fromCharCode(keyPressed); //convert to string
  alert("Key " + character + " was pressed");
}
```

LISTING 6.15 Listener that hears and alerts keypresses

Event	Description
onkeydown	The user is pressing a key (this happens first)
onkeypress	The user presses a key (this happens after onkeydown)
onkeyup	The user releases a key that was down (this happens last)

TABLE 6.8 Keyboard Events in JavaScript

Form events

Forms are the main means by which user input is collected and transmitted to the server. Table 6.9 lists the different form events.

The events triggered by forms allow us to do some timely processing in response to user input. The most common JavaScript listener for forms is the onsubmit event. In the code below we listen for that event on a form with id loginForm. If the pass-
word field (with id pw) is blank, we prevent submitting to the server using preventDefault() and alert the user. Otherwise we do nothing, which allows the default event to happen (submitting the form) as shown in Listing 6.16.

Event	Description
onblur	A form element has lost focus (that is, control has moved to a different element), perhaps due to a click or Tab key press.
onchange	Some <input>, <textarea>, or <select> field had their value change. This could mean the user typed something, or selected a new choice.
onfocus	Complementing the onblur event, this is triggered when an element gets focus (the user clicks in the field or tabs to it).
onreset	HTML forms have the ability to be reset. This event is triggered when that happens.
onselect	When the users selects some text. This is often used to try and prevent copy/paste.
onsubmit	When the form is submitted this event is triggered. We can do some prevalidation when the user submits the form in JavaScript before sending the data on to the server.

TABLE 6.9 Form Events in JavaScript

```
document.getElementById("loginForm").onsubmit = function(e){
  var pass = document.getElementById("pw").value;
  if(pass==""){
    alert ("enter a password");
    e.preventDefault();
  }
}
```

LISTING 6.16 Catching the onsubmit event and validating a password to not be blank

Frame events

Frame events (see Table 6.10) are the events related to the browser frame that con- tains your web page. The most important event is the onload event, which tells us an object is loaded and therefore ready to work with. In fact, every nontrivial event listener you write requires that the HTML be fully loaded.

However, a problem can occur if the JavaScript tries to reference a particular in the HTML page that has not yet been loaded. If the code attempts to set up a listener on this not-yet-loaded , then an error will be triggered. For this reason it is common practice to use the window.onload event to trigger the execution of the rest of the page's scripts.

```
window.onload= function()
{
//all JavaScript initialization here.
}
```

Event	Description
onabort	An object was stopped from loading
onerror	An object or image did not properly load
onload	When a document or object has been loaded
onresize	The document view was resized
onscroll	The document view was scrolled
onunload	The document has unloaded

TABLE 6.10 Frame Events in JavaScript

3.8 Forms

User form input should be validated on both the client side and the server side.

```
<form action='login.php' method='post' id='loginForm'>
  <input type='text' name='username' id='username'/>
  <input type='password' name='password' id='password'/>
  <input type='submit'></input>
</form>
```

LISTING 6.17 A basic HTML form for a login example

To illustrate some form-related JavaScript concepts, consider the simple HTML form depicted in Listing 6.17.

3.8.1 validating Forms

Form validation is one of the most common applications of JavaScript. Writing code to prevalidate forms on the client side will reduce the number of incorrect submis- sions, thereby reducing server load. Although validation must still happen on the server side (in case JavaScript was circumvented), JavaScript prevalidation is a best practice. However, the novice programmer may not be familiar or comfortable using regex, and will often resort to copying a regex from the Internet, without understanding how it works, and therefore, will be unable to determine if it is correct.

Empty Field validation

A common application of a client-side validation is to make sure the user entered something into a field. There's certainly no point sending a request to log in if the username was left blank, so why not prevent the request from working? The way to check for an empty field in JavaScript is to compare a value to both null and the empty string (''') to ensure it is not empty, as shown in Listing 6.18.

```
document.getElementById("loginForm").onsubmit = function(e){
  var fieldValue=document.getElementByID("username").value;
  if(fieldValue==null || fieldValue== ""){
    // the field was empty. Stop form submission
    e.preventDefault();
    // Now tell the user something went wrong
    alert("you must enter a username");
  }
}
```

LISTING 6.18 A simple validation script to check for empty fields

Some additional things to consider are fields like checkboxes, whose value is always set to "on". If you want to ensure a checkbox is ticked, use code like that below.

```
var inputField=document.getElementByID("license");
if (inputField.type=="checkbox"){
 if (inputField.checked)
 //Now we know the box is checked, otherwise it isn't
}
```

Number validation

Number validation can take many forms. You might be asking users for their age for example, and then allow them to type it rather than select it. Unfortunately, no simple functions exist for number validation like one might expect from a full- fledged library. Using parseInt(), isNAN(), and isFinite(), you can write your own number validation function.

Part of the problem is that JavaScript is dynamically typed, so "2" !== 2, but "2"==2. jQuery and a number of programmers have worked extensively on this issue and have come up with the function isNumeric() shown in Listing 6.19. Note: This function will not parse "European" style numbers with commas (i.e., 12.00 vs. 12,00).

```
function isNumeric(n) {
    return !isNaN(parseFloat(n)) && isFinite(n);
}
```

LISTING 6.19 A function to test for a numeric value

3.8.2 submitting Forms

Submitting a form using JavaScript requires having a node variable for the form element. Once the variable, say, formExample is acquired, one can simply call the submit() method:

```
var formExample = document.getElementById("loginForm");
formExample.submit();
```

This is often done in conjunction with calling preventDefault() on the onsub- mit event. This can be used to submit a form when the user did not click the submit button, or to submit forms with no submit buttons at all (say we want to use an image instead). Also, this can allow JavaScript to do some processing before submit- ting a form, perhaps updating some values before transmitting.

It is possible to submit a form multiple times by clicking buttons quickly, which means your server-side scripts should be designed to handle that eventuality. Clicking a submit button twice on a form should not result in a double order, double email, or double account creation, so keep that in mind as you design your applications.