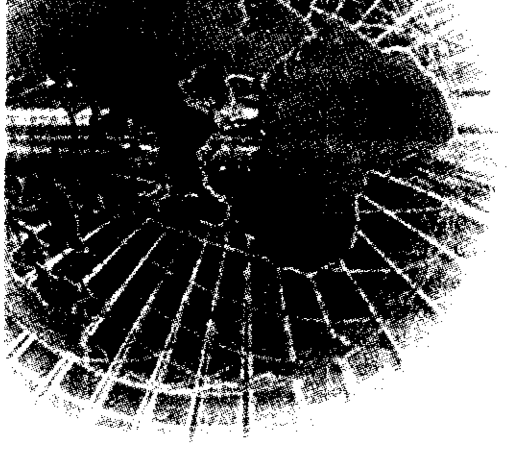


28 AJAX: A New Approach

AJAX, `Asynchronous JavaScript and XML`, is a new technique. Its primary components are `JavaScript` and `XML`. AJAX is a technique, which describes how other technologies, JavaScript, `DOM` (`Document Object Model`), and XML can be used together to create interactive Web applications.

In early days, when we used to create a Web application with these technologies, the applications based on these technologies were known separately. Even the end-user couldn't work on the Web application as desktop-based application. To overcome this, Jesse James Garrett of Adaptive Path combined JavaScript, XML, and DOM together to form a new technique, called `AJAX`. In this technique, the request to the Web server is send by using the `XMLHttpRequest` object. This object, a part of JavaScript technology, helps in sending `asynchronous` request to the server. With this request, Web applications can now interact with Web server `asynchronously`. The time taken to refresh the page also gets minimized which makes the Web application behave like a desktop application. That is why, in AJAX-based Web applications, web pages need not be refreshed repeatedly, when only a part of the page is changing.

In this chapter, we will start with the evolution of Web applications and the technologies used for developing them in bygone days. The focus will next shift to a discussion on the problems associated with these technologies that were used to create Web applications in early days, and how these problems led to the development of AJAX techniques. Last, but not the least, a sample AJAX-based application will make us aware of AJAX benefits.

Evolution of Web Application

In earlier times, applications had their own client-based program that required to be configured on the client machine. Both the server and client needed an upgrade when there were any changes made in the application. However, with the advent of a web-based application, client-server applications changed a lot. A Web application provides a series of web pages to the client, which is accessed by all types of clients using any browser of their choice. There is no need to install a separate client program on all client machines, as the Web browser interprets and displays all web pages and works as a common client for a Web application. Therefore, we can now develop web-based client-server application without spending time on creating separate client programs for different client machines.

In earlier times, we could access simple, static HTML pages using a Web browser, which sends a request to a Web server. The Web server then sends the requested web page, which is stored at the server using HTTP. These web pages are static content, i.e. a constant state: a text file that does not change. When a requirement arose for designing a Web application that processes the data given by the client and presents dynamic content to the client, a problem surfaced. To overcome this problem, an evolution of Web application took place, which led to the development of technologies for processing the client request and generating response content dynamically. With the emergence of this new concept, new technologies also took birth. Thus, the evolution of a Web application gave birth to new technologies, which helped in creating Web applications. The following are a few of the technologies, which can be used to create a Web application:

- Common Gateway Interface (CGI)
- Applets
- JavaScript
- Servlets
- JSP
- ASP
- PHP
- DHTML
- XML

Common Gateway Interface (CGI)

Common Gateway Interface, CGI, is a standard protocol for interfacing the external application software with an information server, commonly known as Web server, as shown in Figure 28.1:

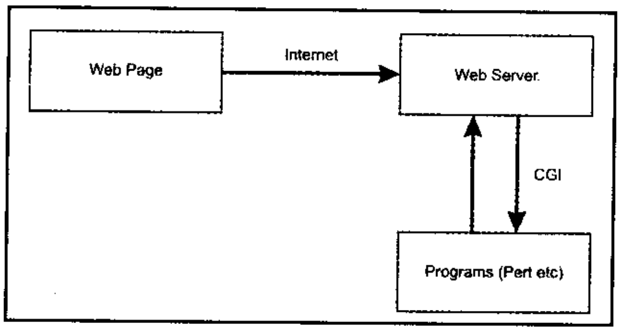


Figure 28.1: Displaying how CGI Works

CGI is not a programming language; rather it is a protocol, which defines a set of rules on how the Web server communicates with the program. This functionality allows the server to pass the request from the client Web browser to the external application. In fact, CGI is a specification used to transfer information between the Web server and the CGI program. A CGI program is written in any programming language, like C, Perl, and Java, etc. CGI programs help the Web server to interact dynamically with web users, e.g. a CGI program is used to process the form's data when it is submitted once. `CGI` also allows `HTML` pages to interact with applications rather than a static web page. Some of the advantages of CGI are as follows:

- It is simple and quick to develop.
- CGI is rich in libraries. Therefore, there is no need to work a lot on your own.

Here are some limitations of CGI:

- CGI is slow; being slow is the flaw of CGI itself and not of the particular programming language used for scripting.
- Each time the process must be launched from the beginning and that takes a lot of time.
- The resources, like the database connections, must be created and reloaded every time.
- The state is not persistent. On every request, everything is built again.

Now let's us move on to another technology called Applets, related to Web applications.

Applets

An applet is a program written in Java programming language, which can be included in the HTML page and run within a Web browser. Java applets are normally used to include small, interactive components to the web page. The applets are mainly used to provide dynamic user-interface and a number of graphical effects for the web pages. When the Web client or the Web browser opens the web page, the applet automatically gets downloaded just like an image. However, with the applet, one cannot control how much of the screen the applet takes up.

JavaScript

Netscape originally invented a simple scripting language called LiveScript. LiveScript was a proprietary add-on to the HTML. When Sun's new programming language, Java, became popular, Netscape quickly switched over and came up with a new scripting language called JavaScript. Only the beginning four alphabets are similar—Java and JavaScript—otherwise both are entirely different from each other.

Normally a Web designer designs web pages and a Web developer codes the application. JavaScript is the scripting language, which is used in many websites by the Web designer to create a client-side application with very less effort. The scripting language is interpreted at runtime and not compiled like other languages, such as C++, C#, etc. Thus, JavaScript is an interpreted language, which means that the scripts execute without compilation. JavaScript is also the client-side language, as it runs on the client browser. JavaScript can be used in almost all the Web browsers, like Internet Explorer, Mozilla, Firefox, Netscape, etc. and it can easily interact with the HTML elements.

Uses of JavaScript

Basically, JavaScript is designed to create interactivity with HTML pages. The following are the uses of JavaScript:

- JavaScript is simpler, so anyone can put the small snippets of the code into their HTML pages.
- JavaScript enables writing of dynamic text into HTML page. The variable text can also be written in HTML page, e.g. `document.write("<h1>" +name + "</h1>")`. This command will write the text of the name variable into the HTML page.
- JavaScript enables you to read and change the content of HTML controls. For example, the text inserted in the text field of an HTML page can be read with the help of JavaScript.
- Certain validations to be performed on the client-side, such as not leaving any text field blank, match of the password and the confirmation of password fields, etc. can be checked at client-side by using JavaScript as the scripting language.
- JavaScript is also helpful in creating cookies. It can be used to either store or retrieve relevant information on the client's computer.
- JavaScript also enables you to load a specific page depending upon the client's request.
- JavaScript is used to write functions that are embedded in or included from HTML pages and interact with the Document Object Model (DOM) of the page.
- JavaScript is also helpful in changing the image as the mouse cursor moves over them.
- JavaScript is also helpful in calling the new web page, according to the client or user's action

Till now, you were aware of the basic technologies related to Web application programming. Now let's understand the technologies required at the time of Web application development.

Servlets

Java Servlets are an alternative to CGI programs. The major difference between CGI and Servlets is that the Servlets are persistent. In other words, once loaded it stays to fulfill other subsequent requests. On the contrary, CGI disappears after fulfilling the request once. Moreover, similar to the applets that run on the browser, the Servlets run on Java-enabled Web server.

The Java Servlet API allows the developer to add dynamic content to a Web server using Java platform. The Servlets maintain the state across multiple requests by using HTTP cookies, session variables, URL rewriting or the Hidden fields. The Servlet API contains `javax.servlet` package hierarchy. The Servlet API also specifies the expected interaction of the Web container and a Servlet. The Web container is part of the Web server and it performs many tasks. Some of these tasks are interaction with the Servlets, managing the life cycle of the Servlets, and mapping the URL to a particular Servlet, if the URL requester has the correct access rights.

The main purpose of the Servlets is to serve HTML to the client, mainly through HTTP protocol. Servlets are created, managed, and destroyed by the Web server, where they run. The Servlets are recognized in the context of the Web server.

JavaServer Pages (JSP)

JavaServer Pages (JSP) is an enhancement to the Java Servlet technology from Sun Microsystems. The JSP technology provides a technique to dynamically generate web pages. JSP also simplifies the process of creating or developing web-based applications.

The JSP technology allows HTML to be combined with Java on the same page. In response to the Web client's request, JSP allows software developers to dynamically generate HTML documents. The JSP technology allows Java code and certain pre-defined actions to be embedded into static content. The Java code is added with the use of the JSP directives, JSP scripting elements, and JSP actions. The JSP syntax adds additional XML-like tags, called JSP actions. The JSP compiler compiles JSP into Servlets. The JSP compiler can directly generate byte code for the Servlets or can generate a Servlet in Java code, which is then compiled by the Java compiler. The extension of the JSP files is (`.jsp`).

Figure 28.2 shows the different clients connecting to the Web server through internet:

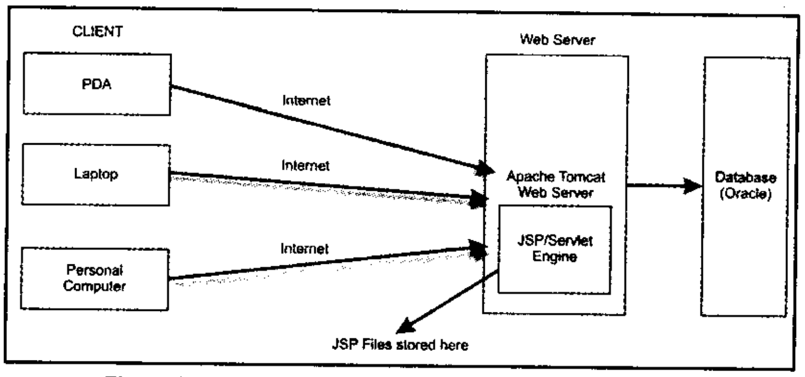


Figure 28.2: Different Clients Connected to the Web Server

In Figure 28.2, the most popularly used Web server, Apache Tomcat Web server, is running on Windows. As shown in the Figure 28.2, the JSP files run on the Web server in the JSP Servlet engine. The JSP Servlet engine dynamically generates the HTML output and sends it to the client's browser.

Active Server Pages (ASP)

Active Server Pages (ASP) is the Microsoft's server-side scripting engine for dynamically generating HTML or the web pages for the Web browser. The default scripting language used to write ASP is VBScript. The simple approach to understand ASP is that ASP is a program that runs inside IIS. IIS stands for Internet Information Server. In competition to JSP, a server-side technology given by Sun Microsystems, ASP is the server-side technology provided by Microsoft.

The ASP page though similar to the HTML page, also contains text, XML, and scripts. The scripts in an ASP file are executed on the server. The extension of the ASP files is (`.asp`). When the Web browser sends the request for an ASP file, the IIS passes the request to the ASP engine and then, after reading it line by line, the ASP engine executes the scripts in the file. Later, in the plain HTML format, the ASP file is returned to the Web browser.

The modification and additions to the contents of the web page can be done dynamically with the help of ASP. The data from the database can also be accessed with the help of ASP. In addition, the result is returned to the Web browser. ASP, in comparison to CGI and Perl, is simpler and faster in speed.

PHP

Till now, you were aware of Sun Microsystems and Microsoft technologies, which are JSP and ASP, respectively. Let's move on to understand another server-side scripting technology called PHP, which is an alternative to ASP and JSP. PHP helps to create dynamic web pages. You can embed PHP into HTML pages and generate a dynamic web page. PHP programs can be deployed on a Web server. PHP uses the concept of CGI for server-side programming. It acts as a filter for displaying the dynamic content and can be used for extracting data from a database.

Dynamic HTML (DHTML)

Before studying DHTML in detail, let's first expand the term DHTML. DHTML stands for Dynamic Hypertext Markup Language and is the art of making the HTML pages dynamic. DHTML is a combination of technologies used to create dynamic and interactive websites and Web applications. Initially in standard HTML, once the page is loaded from the server it will not change until another request to the server. On the contrary, dynamic HTML provides much control over HTML elements. It allows the HTML elements to change at any time without returning to the Web server and uses the DOM, CSS, HTML, and JavaScript to develop interactive Web applications.

Document Object Model (DOM)

DOM allows changing any part of the web page using DHTML. DOM is the API that serves as glue for binding a scripting language, like JavaScript, with the markup language, like HTML. HTML DOM defines the standard set of objects for HTML and allows access and manipulation of HTML objects in a standard way. It helps in specifying each part of the web page and allows access by using naming conventions. A detailed study about DOM is covered in the next chapter, which discusses the different levels of DOM that serve as the DOM specification and more.

Cascading Style Sheets (CSS)

The CSS is used in DHTML to control the look and feel of a web page. CSS is used to style the HTML elements. The font color, font text, background color, images, and the placements of objects on the web page are defined in the Cascading Style Sheets. CSS allows the developer to control the style and layout of various web pages. This can be done by defining the style for each HTML element and then can be applied to multiple web pages at a time. This also enables a quick global change. In other words, if the same style is applied to all the HTML elements then, on simply making changes in the style, all the elements will get updated automatically. Therefore, DHTML helps in making the web pages dynamic and provides a good look and feel of the web page with the help of CSS. The HTML elements or objects are placed in the web page by using XHTML. With the help of DOM specified in the web page, these objects, placed in the web page, can be accessed or manipulated at any time. Now let's understand XML.

XML

XML is the extensible markup language used for the exchange of information between the applications or the organizations. XML allows the designers to create their own user-defined tags and enable the transmission, validation, and interpretation of data between applications or the data between the organizations. In simpler words, it allows the designer to provide data in tags by creating meaningful tags. Some of the XML-related technologies are as follows:

- XHTML
- XML DOM
- XSLT
- XML Parser

Let's have an overlook to the listed XML-technologies.

XHTML

As discussed in DHTML, XHTML is a cleaner but stricter version of HTML. XHTML refers to the eXtensible Hypertext Markup Language whose focus of existence is to replace HTML. XHTML is similar to HTML 4.x and is defined in the form of a XML application. It consists of elements in HTML 4.01, combined with the syntax of XML. As previously stated, XML is the markup language which results in well-formatted documents. So, XHTML provides the privilege of writing well-formed documents, which work in all the Web browsers. Certain rules to be followed while using XHTML are as follows:

- XHTML elements should be properly nested
- XHTML elements should always be closed
- XHTML elements should be in lowercase
- XHTML documents must have a single root element

XML DOM

DOM refers to the Document Object Model and presents the XML document as the tree-structure having the Root node as the parent element and the Elements, Attributes, and Text defined as the child nodes. So, XML DOM defines the standard way for accessing and manipulating XML documents. The elements containing the text and the attributes, with the help of the DOM tree, can be manipulated and accessed. The contents of these elements can be modified, new elements can be created, or the unwanted elements can be removed from the DOM tree. The most important thing to be noted is that all the Elements, their Text, and their Attributes are known as the nodes.

In the DOM structure, the entire document is considered as the Document node; XML tag or the XML element is recognized as the Element node; the text in the XML elements are referred to as the Text node; attributes are considered the Attribute nodes; and the comments are considered the Comment node. In the DOM tree-structure, the nodes have a hierarchical relationship with each other. The terms parent and child are used to describe the relationships between the nodes. Let's consider an example of a XML file and look at its DOM tree-structure.

The code given in Listing 28.1shows code for `product.xml` file containing the data related to various products:

Listing 28.1: products.xml

```
<? xml version="1.0" encoding="UTF-8"?>
<PRODUCTDATA>
  <PRODUCT PRODID="P001">
        <PRODUCTNAME>Barbie Doll</PRODUCTNAME>
<DESCRIPTION>This is a toy for children in the age group below 5 years  </DESCRIPTION>
        <PRICE>$24.00</PRICE>
        <QUANTITY>12</QUANTITY>
  </PRODUCT>
  <PRODUCT PRODID="P002">
        <PRODUCTNAME>Mini Bus</PRODUCTNAME>
<DESCRIPTION>This is a toy for children in the age group of 5-10 years  </DESCRIPTION>
        <PRICE>$42.00</PRICE>
        <QUANTITY>6</QUANTITY>
  </PRODUCT>
  <PRODUCT PRODID="P003">
        <PRODUCTNAME>Car</PRODUCTNAME>
<DESCRIPTION>This is a toy for children in the age group of 10-15 years </DESCRIPTION>
        <PRICE>$60.00</PRICE>
        <QUANTITY>21</QUANTITY>
  </PRODUCT>
</PRODUCTDATA>
```

In Listing 28.1, `<PRODUCTDATA>` is the root element of the document. Since all the other elements are within the `<PRODUCTDATA>` element, it is considered as the root element. The root element has three `<PRODUCT>` nodes and an Attribute node named, `PRODID`. Each of the Element nodes has a Text node as well.

Figure 28.3 shows the DOM tree-structure for `products.xml`:

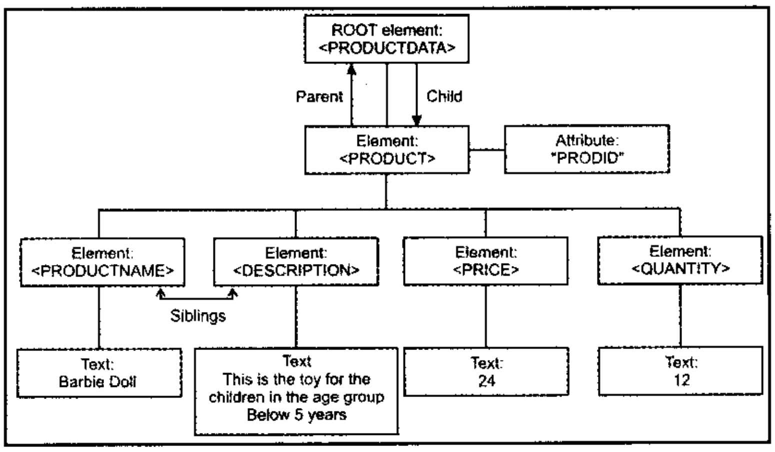


Figure 28.3: Showing DOM node Tree-Structure

Figure 28.3 shows only one child node <PRODUCT> of the parent node <PRODUCTDATA>. The DOM node tree-structure shown earlier has the Root node `<PRODUCTDATA>` containing the three child nodes, named `<PRODUCT>`. Each `<PRODUCT>` child node has four Element nodes and an Attribute node. Each Element node has the respective Text node, as shown in the Figure 28.3. We will go into detail about the various levels of the DOM and how you can navigate through the nodes in the next chapter.

XSLT

XSLT is the language used for transforming XML documents into XHTML or other XML documents. XSLT uses XPath for navigating through XML documents and finding information in it. In the transformation process, XSLT uses the XPath searches for those parts of the source document, which match the pre-defined template. When a match is found, XSLT transforms the source document into the resultant document by applying the pre-defined template.

XML Parser

The XML parser is used to read, update, create, and manipulate XML documents. For manipulating the XML document, the XML parser loads the document into the computer's memory and then manipulates data by using the DOM node-tree-structure. The XML parser is the part of the software, which reads the XML files and tests whether the XML document is well-formed against the given DTD or the XML schema. Moreover, the XML parser also makes the XML files available to the application with the use of the DOM.

Till now the chapter dealt with the explanation of almost all the technologies used to create Web applications. All these technologies discussed earlier share a common problem and AJAX proves to be the solution for these problems. Read on to know about them.

Problems with Technologies

All the previously mentioned technologies use the Classical or traditional Web application model. In the Classical or the traditional Web application model, the nature of interaction between the client and the server is of start-stop-start-stop. In a traditional Web application model, the browser responds to the user action by discarding the current HTML page. Then the request is sent back to the Web server and when the server completes the processing of request, it returns the response page to the Web browser. Finally, the browser refreshes the screen and displays the new HTML page. You will be surprised to note that the user is bound to not do anything, until the entire process completes.

In technical terms, the problem with the classical Web applications model was the synchronous request-response communication model. This can be explained with the help of Figure 28.4:

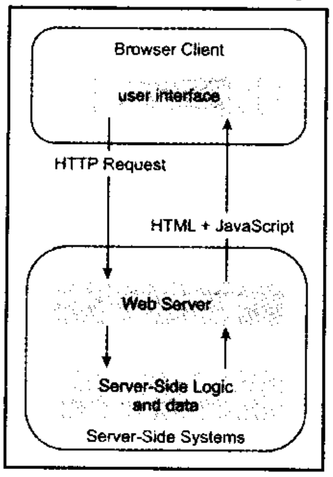


Figure 28.4: Classical Web Application Model

This Classical Web Application model, shown in Figure 28.4, makes technical sense; however it does not provide the best user experience. Since this classical application model keeps the user waiting, it does not provide the best user experience. To overcome this, the developer noticed a technical approach that Google used. After analyzing the facts of Google, on February 18, 2005 Jesse James Garrett, President, and founder of the Adaptive Path came out with a new technique AJAX—that was based on the approach used by Google.

Let us move on further to learn about AJAX.

AJAX—The Solution

AJAX, a new approach to Web applications, is based on several technologies that help to develop applications with better user experience. It uses JavaScript and XML as the main technology for developing interactive Web applications. These applications are based on AJAX Web application model, which uses JavaScript and XMLHttpRequest object for asynchronous data exchange. The JavaScript uses XMLHttpRequest object to exchange data asynchronously over the client and server. Let's move further to have a detail study on the AJAX Web application Model.

AJAX Web Application Model

You already know that the major issue, with regard to the Classical Web application model, was resolved through AJAX. The AJAX application eradicates the start-stop-start-stop nature or the click, wait, and refresh criteria of the client-server interaction. Figure28.5 shows how the intermediary layer is introduced between the user and the Web server:

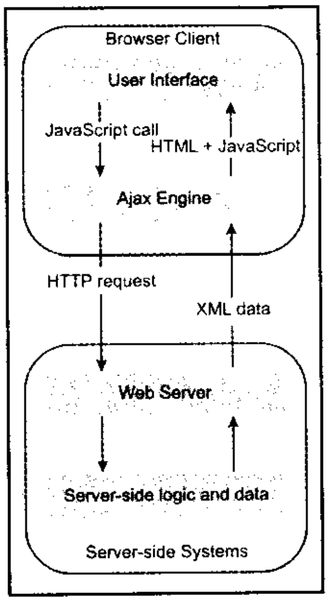


Figure 28.5: AJAX Web Application Model

Instead of loading the web page during the beginning of the session, the browser loads the ASP engine, written in JavaScript. As shown in Figure 28.5, the web page sends its requests using a JavaScript function. This JavaScript code makes a request to the server. The server response comprises of data and not the presentation, which implies that the data required by the page is provided by the server as the response, and the style or presentation is implemented on that data with the help of the markup language. Most of the page does not change. Only parts of the page that need to change are updated. The JavaScript dynamically updates the web page, without redrawing everything. For the Web server, nothing has changed; it still responds to each request, just as it did before.

Though JavaScript makes a request to the server, you can still type in Web forms and even click buttons, while the Web server is still working in the background. Then, when the server completes its processing, your code updates just the part of the page that has changed. This way you never have to wait around. That is the power of asynchronous requests. AJAX engine, between the user and the application, irrespective of the server, does asynchronous communication. This prevents the user from waiting for the server to complete its processing. The AJAX engine takes care of displaying the user interface and the interaction with the server on the user's behalf.

But in traditional Web applications, the synchronous mode of communication existed between the client and the server, as shown in the Figure 28.6:

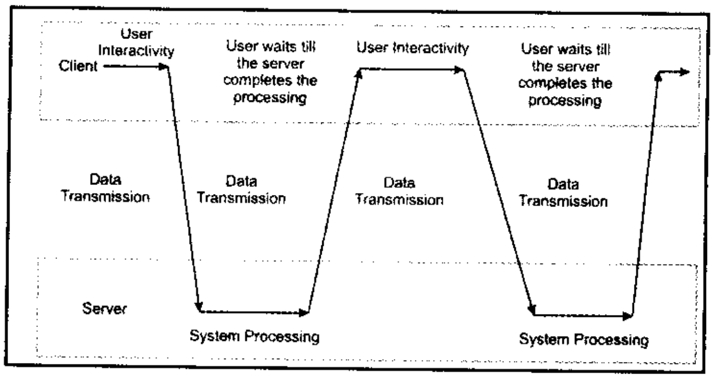


Figure 28.6: Synchronous Mode of Communication

Since the essence of AJAX is a partial screen update and the asynchronous communication, the programming model, shown in the Figure 28.7, is not bound to a specific data exchange format or the specific programming language or the specific communication mechanism:

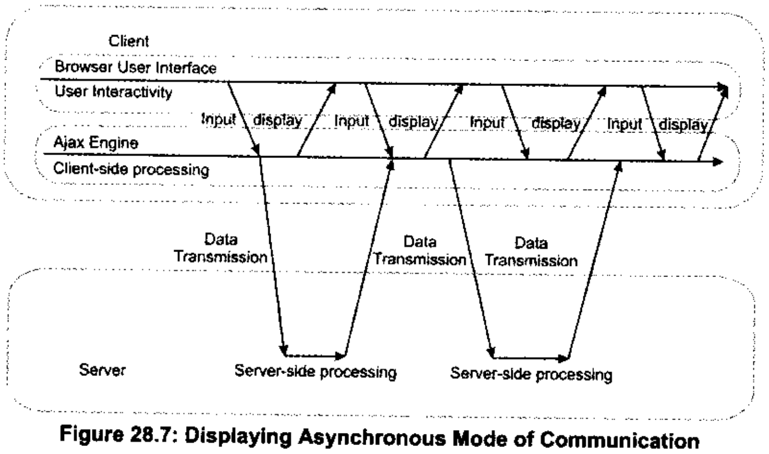


Figure 28.7: Displaying Asynchronous Mode of Communication

As shown in Figure 28.7, every user action generates an HTTP request that takes the form of a JavaScript to call the AJAX engine. Any response to the user action does not require the trip back to the server as in the Classical Web application model. Rather, the AJAX engine handles on its own, i.e. the data validations, some navigation, editing data in memory, are handled by the AJAX engine.

If the AJAX engine needs something from the server, like retrieving new data or loading additional interface code, then the engine makes the asynchronous interaction with the server, using JavaScript and XMLHttpRequest object for asynchronous data exchange. The engine's interaction with the server does not interrupt the user's interaction with the application. In this way, the asynchronous communication is done with the help of the AJAX engine.

NOTE

For more information on the XMLHttpRequest Object, refer Chapter 30.

Comparing Figures 28.6 and 28.7, it is seen that in the asynchronous mode of interaction, there is no scope for the user to wait until the server-side processing gets over. The AJAX Web application model allows users to continue working, and simultaneously, if necessary, the AJAX engine interacts with the server without interrupting the user's interaction with the application.

After learning how AJAX works and how the problems or shortcomings of traditional technologies are overcome by using AJAX, let's create the first AJAX application by using JavaScript.

Creating a Sample AJAX Application

Let's consider a scenario of a Jewelry showroom. The owner wants to design a web page that would display the different jewelry items along with some information to its various customers. When the user points to an image whose information he wants to know, the details will be displayed on the web page without the page being refreshed repeatedly.. In this application, we first need to create a Jewelery.html page, which shows information about the different jewelry in the showroom.

The code, given in Listing 28.2, shows code for the Jewelry.html file of the application (you can find Jewelry.html file in the Code\AJAX\Chapter 28\Jewelry folder on the CD):

Listing 28.2: Jewelry.html

```
<html>
<head>
<title>First AJAX Application</title>
<script language = "javascript">
  var XMLHttpRequestObj = false;

  if (window.XMLHttpRequest) {
        XMLHttpRequestObj = new XMLHttpRequest();
  } else if (window.ActiveXObject) {
        XMLHttpRequestObj = new
        ActiveXObject("Microsoft.XMLHTTP");
  }

  function getData(dataSource, divID)
  {
        if(XMLHttpRequestObj) {
              var obj = document.getElementById(divID);
              XMLHttpRequestObj.open("GET", dataSource);

              XMLHttpRequestObj.onreadystatechange = function()
              {
                    if (XMLHttpRequestObj.readyState == 4 &&
                    XMLHttpRequestObj.status == 200) {
                          obj.innerHTML =
                          XMLHttpRequestObj.responseText;
                    }
              }
              XMLHttpRequestObj.send(null);
        }
  }
</script>
</head>
<body>
  <H1>First Application using AJAX</H1>
  <img src="image1.jpg" onmouseover = "getData('bangles.txt','targetDiv')">
  <img src="image2.jpg" onmouseover = "getData('rings.txt','targetDiv')">
  <img src="image3.jpg" onmouseover = "getData('necklaces.txt','targetDiv')">
  <div id="targetDiv">
        <h1>Welcome to my Jewelery Showroom!</h1>
  </div>
</body>
</HTML>
```

In Listing 28.2, an HTML page is designed displaying the images of the various jewelry items available in the showroom. As soon as the user points the mouse pointer on any of the three images, the text saved in the respective text files are displayed on the Web browser. The three text files are as follows:

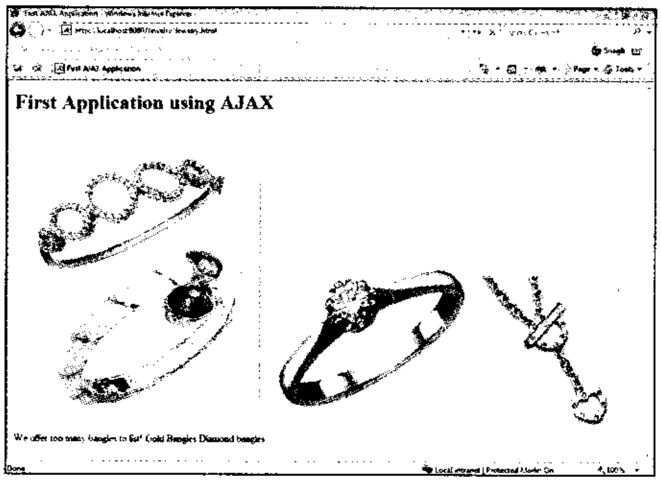
- bangles.txt
- rings.txt
- necklaces.txt

To run the Jewelery application, ensure that a Web server is configured on your machine. Here, we are using Tomcat as a Web server for running AJAX-based Web application. Follow these steps to run the application:

- Open the Internet Explorer (IE).
- Type the following address (http://localhost:8080/Jewelry/Jewelry.html) in the address bar of the IE and press enter to open the page (Figure 28.8).

NOTE

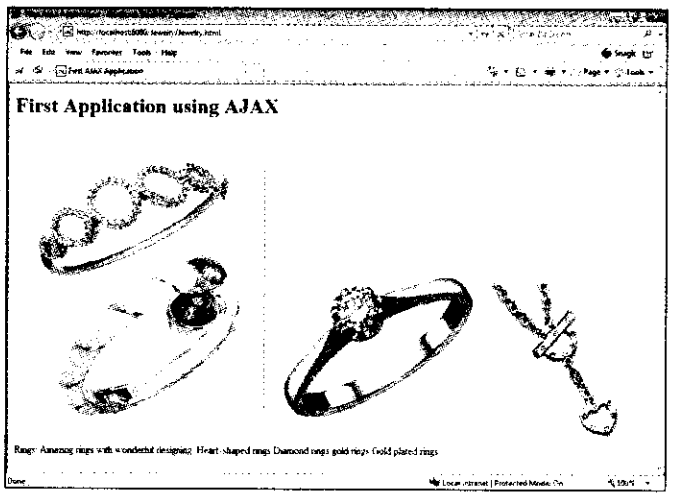
As we are using Tomcat web-server to deploy all the applications given in this book, the port used in our case is 8080. It can vary in your case. The default port used by Tomcat web-server is 8080.


First Application using AJAX

We offer too many bangles to list! Gold Bangles Diamond bangles

Figure 28.8: Showing Output of the Jewelry Application

- When you move the mouse pointer on any of the images, the text related to that image get displayed, as shown in the Figure 28.9:


First Application using AJAX

Rings: Amazing rings with wonderful designing Heart-shaped rings Diamond rings gold rings Gold plated rings

Figure 28.9: Displaying a Simple AJAX Example

As the mouse moves from one image to another, the JavaScript in the page fetches some new text and replaces the older text, without even a screen flicker or page fetch or fuss.

Now let's understand how the Jewelery.html page uses AJAX. When the mouse moves over any of the three images, the `onmouseover` event is generated and the JavaScript method, `getData`, is called as shown in the following code snippet:

```
<body>

  <H1>First Application using AJAX</H1>
  <img src="Image1.jpg" onmouseover="getData('bangles.txt','targetDiv')">
  <img src="Image2.jpg" onmouseover="getData('rings.txt','targetDiv')">
  <img src="Image3.jpg" onmouseover="getData('necklaces.txt','targetDiv')">

  <div id="targetDiv">
        <h1>Welcome to my Jewelery Showroom!</h1>
  </div>
</body>
```

The `getData` method passes two text strings—firstly, the name of the text file, like `bangles.txt` or `rings.txt` or `necklaces.txt`, and secondly the name of the `<div>` element.

Listing 28.3 shows code for bangles.txt file (you can find bangles.txt file in the Code\AJAX\Chapter 28\Jewelry folder on the CD):

Listing 28.3: bangles.txt

```
We offer too many bangles to list!
Gold Bangles
Diamond bangles
```

Listing 28.4 shows code for the rings.txt file (you can find rings.txt file in the Code\AJAX\Chapter 28\Jewelry folder on the CD):

Listing 28.4: `rings.txt`

```
Rings: Amazing rings with wonderful designing.
Heart-shaped rings
Diamond rings
gold rings
Gold plated rings
```

Listing 28.5 shows code for the `necklaces.txt` (you can find necklaces.txt file in the Code\AJAX\Chapter 28\Jewelry folder on the CD):

Listing 28.5: `necklaces.txt`

```
The all kind of diamond and gold necklaces are also available.
Diamond necklace
Gold necklace
The necklaces are also designed according to your choice.
```

Apart from the preceding text files, three image files of the bangles, rings, and necklaces are also displayed on the web page.

Any of these three texts is downloaded by the browser from the server, which is in the background, while the user is working with the rest of the web page. Read on further to understand how this is done.

*Creating the **XMLHttpRequest** Object*

The application created here requires the `XMLHttpRequest` object. In Listing 28.2, towards the beginning of Jewelery.html code, locate the following code snippet:

```
<script language = "javascript">
var XMLHttpRequestObj = false;
.
.
.
```

This code snippet declares a variable XMLHttpRequestObj. Since this code is outside any function, it runs immediately when the page loads. This variable is set to false, so that the script can check later whether the variable is created or not. In case of browsers, like Netscape, FireFox, Opera, XMLHttpRequest object is usually the part of the browser's window object and it can be accessed as window.XMLHttpRequest. If the window.XMLHttpRequest returns true then a XMLHttpRequest object is created with the following code:

```
if (window.XMLHttpRequest) {
    XMLHttpRequestObj = new XMLHttpRequest();
}
```

On the contrary, a different perspective is required for the Internet Explorer Web browser. The ActiveX object in the Internet Explorer (version 5 and so on) is used to create the XMLHttpRequest object as shown here:

```
if (window.ActiveXObject) {
    XMLHttpRequestObj = new
    ActiveXObject ("Microsoft.XMLHTTP");
}
```

So, depending upon the browser you are using, an XMLHttpRequest object is created.

When the user moves the mouse over the images, an "onmouseover" event is generated which calls the getData () function. When the getData () function is called, the XMLHttpRequest object is first checked to see whether it is valid, and then further processing is done.

Opening **XMLHttpRequest** Object for Asynchronous Downloads

When the valid object of the XMLHttpRequest is created, the object calls its open () method. You can configure the object to use the URL you want by using the object's open method. The syntax of open () method is as follows:

```
req.open("Get",URL,true);
```

Here, the first parameter indicates the type of HTTP method that is used for sending request; the second parameter is the URL of the requested resource and; the third parameter is optional, which shows whether the request is synchronous or Asynchronous. The default value of third parameter is true, which indicates an Asynchronous request.

Here, in this application, the URL from which the data you want to fetch is passed from the getData function as the dataSource argument. The URL can be opened with the standard HTTP techniques, like GET or POST or PUT. This example uses the GET method to request the respective text file on the server:

```
XMLHttpRequest.open("GET", dataSource);
```

After opening the XMLHttpRequest object, the XMLHttpRequest object has the property named onreadystatechange, which allows handling the asynchronous loading operations. If this property is assigned to the name of any JavaScript function, then this function will be called each time the XMLHttpRequest object's state changes.

This JavaScript function is also known as "Callback" function. When the server returns with the information, the callback function is invoked. In turn, the callback function can display the new information to the user. We defined the callback function with the following JavaScript code:

```
XMLHttpRequestObj.onreadystatechange = function ()

{
    if (XMLHttpRequestObj.readyState == 4 &&
     XMLHttpRequestObj.status == 200) {
          obj.innerHTML =
          XMLHttpRequestObj.responseText;
    }
}
```

Finally, when the XMLHttpRequest object is in its ready state and the status is equal to 200, then the data is fetched. The readyState value "0" indicates that the request is completed and the status 200 refers to the 'Ok' state of the XMLHttpRequest object, which means that the request resource is completely downloaded. The five ready states of the XMLHttpRequest object are as follows:

- 0 for uninitialized state
- 1 for loading state
- 2 for loaded state
- 3 for interactive state
- 4 for the complete state

The status property holds the status of the download. Table 28.1 lists some possible values of the status property:

Table 28.1: Possible values for the status property of XMLHttpRequest object

Status	Possible values
Ok	200
Created	201
No Content	204
Reset Content	205
Partial Content	206
Bad request	400
Unauthorized status	401
Forbidden status	403
Not Found status	404
Method Not Allowed	405
Not Acceptable	406
Proxy Authentication Required	407
Request Timeout	408
Length Required	411
Requested Entity Too Large	413
Requested URL Too Long	414
Unsupported Media Type	415
Internal Server Error	500
Not Implemented	501
Bad Gateway	502
Service Unavailable	503
Gateway Timeout	504
HTTP Version Not Supported	505

So, to make sure that the data is completely downloaded, the value for the status property must be 200. Finally, when the data is completely downloaded, it is retrieved in either the standard HTML or the XML format. Here, the `responseText` property is used. So the data is retrieved in standard HTML format. However, if your data is formatted as XML, then `responseXML` property is used.

After retrieving the data, you have to display the data on the web page. The data is displayed by using the HTML <div> element. This is done as follows:

```
<div id="targetDiv">
<h1>Welcome to my Jewelery Showroom!</h1>
</div>
```

The <div> element shows the location where you want to display the data. The id attributes is used to identify the <div> element and it is passed as an argument to getData () function as shown here for bangles.txt file.

```
getData ('bangles.txt','targetDiv')
```

Finally, we have created and understood the simple AJAX application, considering the Jewelery showroom scenario.

Summary

In this chapter, the discussion began with the evolution of Web applications, along with the various technologies, like CGI, Applets, JavaScript, Servlets, JSP, DHTML, and XML used for creating web applications. Next we discussed the problems related with Classical Web Application model, which is based on the synchronous mode of interaction of the client with the server. Due to this, the users had to wait a lot till the entire processing of the server completed. Next we learnt about the AJAX Web Application model and how it solved the problem of Classical Web Application model. When the AJAX Web Application model was compared with the Classical Web application model, it showed us that AJAX Web application model used AJAX engine, which helps in the asynchronous mode of interaction between the client and the server.

After going through all these theoretical concepts, we proceeded towards creating a sample AJAX application, which build the concept on how the `XMLHttpRequest` object is created and used.

In the next chapter, we will discuss the basics of JavaScript that is an essential component of AJAX.

Quick Revise

Q1. Define AJAX.

Ans: Asynchronous JavaScript and XML (AJAX), is a new technique of web programming. Its primary components are JavaScript and XML. AJAX is a technique, which describes how other technologies, JavaScript, DOM (Document Object Model), and XML can be used together to create interactive Web applications.

Q2. Define CGI.

Ans: Common Gateway Interface (CGI) is a standard protocol for interfacing the external application software with an information server, commonly known as Web server. CGI is not a programming language; rather it is a protocol, which defines a set of rules on how the Web server communicates with the program. This functionality allows the server to pass the request from the client Web browser to the external application. In fact, CGI is a specification used to transfer information between the Web server and the CGI program. A CGI program is written in any programming language, like C, Perl, and Java, etc.

Q3. List the usage of JavaScript.

Ans: The following are the uses of JavaScript:

- JavaScript is simpler, so anyone can put the small snippets of the code into their HTML pages.
- JavaScript enables writing of dynamic text into HTML page. The variable text can also be written in HTML page, e.g. `document.write("<h1>" +name + "</h1>")`. This command will write the text of the name variable into the HTML page.
- JavaScript enables you to read and change the content of HTML controls. For example, the text inserted in the text field of an HTML page can be read with the help of JavaScript.
- Certain validations to be performed on the client-side, such as not leaving any text field blank, match of the password and the confirmation of password fields, etc. can be checked at client-side by using JavaScript as the scripting language.

- JavaScript is also helpful in creating cookies. It can be used to either store or retrieve relevant information on the client's computer.
- JavaScript also enables you to load a specific page depending upon the client's request.
- JavaScript is used to write functions that are embedded in or included from HTML pages and interact with the Document Object Model (DOM) of the page.
- JavaScript is also helpful in changing the image as the mouse cursor moves over them.
- JavaScript is also helpful in calling the new web page, according to the client or user's action

Q4. **Define JSP.**

Ans: JavaServer Pages (JSP) is an enhancement to the Java Servlet technology from Sun Microsystems. The JSP technology provides a technique to dynamically generate web pages. JSP also simplifies the process of creating or developing web-based applications.

Q5. **Define DOM.**

Ans: Document Object Model (DOM) is the API that serves as glue for binding a scripting language, like JavaScript, with the markup language, like HTML. HTML DOM defines the standard set of objects for HTML and allows access and manipulation of HTML objects in a standard way. It helps in specifying each part of the web page and allows access by using naming conventions. A detailed study about DOM is covered in the next chapter, which discusses the different levels of DOM that serve as the DOM specification and more.

Q6. **Define XML.**

Ans: Extensible Markup Language (XML) is the language used for the exchange of information between the applications or the organizations. XML allows the designers to create their own user-defined tags and enable the transmission, validation, and interpretation of data between applications or the data between the organizations

Q7. **What are the usage of XSLT?**

Ans: Extensible Markup Language Transformation (XSLT) is the language used for transforming XML documents into XHTML or other XML documents. XSLT uses XPath for navigating through XML documents and finding information in it. In the transformation process, XSLT uses the XPath searches for those parts of the source document, which match the pre-defined template. When a match is found, XSLT transforms the source document into the resultant document by applying the pre-defined template.

Q8. **Explain the AJAX Web application model.**

Ans: The AJAX application eradicates the start-stop-start-stop nature or the click, wait, and refresh criteria of the client-server interaction. Instead of loading the web page during the beginning of the session, the browser loads the ASP engine, written in JavaScript, the web page sends its requests using a JavaScript function. This JavaScript code makes a request to the server. The server response comprises of data and not the presentation, which implies that the data required by the page is provided by the server as the response, and the style or presentation is implemented on that data with the help of the markup language. Most of the page does not change. Only parts of the page that need to change are updated. The JavaScript dynamically updates the web page, without redrawing everything. For the Web server, nothing has changed; it still responds to each request, just as it did before.

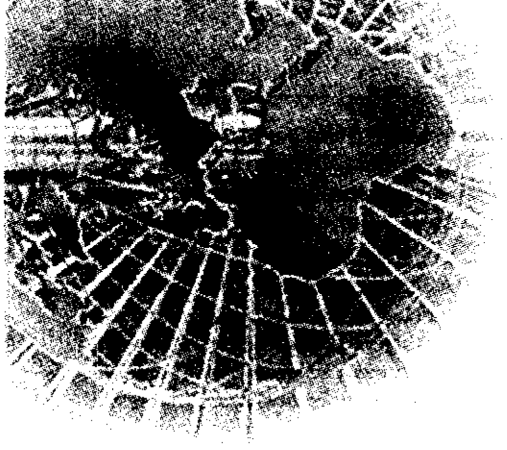
Q9. **How the AJAX engine communicates with the server?**

Ans: When the AJAX engine needs something from the server, like retrieving new data or loading additional interface code, then the engine makes the asynchronous interaction with the server, using JavaScript and XMLHttpRequest object for asynchronous data exchange. The engine's interaction with the server does not interrupt the user's interaction with the application. In this way, the asynchronous communication is done with the help of the AJAX engine.

Q10. List the noteworthy status properties of AJAX.

Ans: Following are the noteworthy status properties of AJAX:

i. Ok	200
ii. Create	201
iii. No Content	204
iv. Bad Request	400
v. Method Not Allowed	405
vi. Bad Gateway	502
vii. HTTP Version Not Supported	505



29

Understanding JavaScript for AJAX

JavaScript was developed by Brendan Eich and introduced in the 1995 release of the Netscape 2.0. It was standardized by the European Computer Manufacturer's Association (ECMA) as ECMAScript. The ECMAScript standard defines the core of the `JavaScript` language. Most browsers today support the third edition of the ECMA-262 standard. JavaScript adds interactivity to web pages made in HTML and this made `JavaScript` more popular and widely used very quickly.

`JavaScript` is an essential part of AJAX. The `JavaScript` code in an AJAX application sends the requests of the client to be processed by the server, but doesn't wait for an answer. Even better, `JavaScript` can also work with the server's response that comes in the form of XML, instead of reloading the entire page when the server is finished with your request. In this case, the `JavaScript` processes the XML document according to `DOM` model. It serves as the intermediary between the browser and the server so that a web page can be updated dynamically without refreshing the entire page.

Introduction to **JavaScript**

With the advent of the Web, HTML and CGI (Common Gateway Interface) were the only two technologies available for developing the web pages. HTML defined the part of the text document and instructed the browser to display it. However, HTML has one disadvantage of being static. If you wish to change something or want to use the data entered by the user, you need to make a round trip to the server. This procedure, indeed, took a lot of time.

When dynamic technologies, such as ASP, ASP.NET, PHP, or JSP, arrived in the market, we got an ability to create dynamic web pages, which had the ability to interact with the user. In this case, the user entered some data into the forms, which was then sent to the server. The server processed the data and then provided the response to the browser in the form of HTML document. The problem with this type of approach was that each time there was a change; the whole process had to be repeated. This was cumbersome, slow, and not as impressive as the new media 'Internet' promises to be. The display of page meant the reloading of page, which was a slow process and failed frequently.

Now there arose the need for a new technology, which would allow the Web developers to develop Web applications that provide immediate feedback to the users and change the HTML without reloading the page from the server every time. Suppose there is a form, which is reloaded every time there is an error in any of the fields of the form. This would be quite a slow process. But, we can make it a fast process by using `JavaScript` in your application. This is done by JavaScript at the client side, which flag an error message as soon as the error occurs in the form without reloading the page and without moving to the Web server.

`JavaScript` is executed by the browser on the user's computer. This type of code is called client-side code and this type of approach results in fast-running web sites. It is a scripting language—a system of programming codes—created by Netscape and can be embedded into HTML of a web page in order to add more interactivity and functionality. `JavaScript` is an interpreted language, i.e. it doesn't need any compiler to execute a code.

When `JavaScript` was first introduced, its name was LiveScript, but Netscape changed the name to `JavaScript`. JavaScript is a scripting language and it is not related to Java in any way. Netscape included `JavaScript` in their Netscape Navigator 2.0 browser via an interpreter, which read and executed the `JavaScript` code added to `.html` pages. In general, script languages, such as `JavaScript`, are faster and easier than the structured languages, such as C++ and Java.

In its long journey, `JavaScript` has continuously grown in popularity and today it is supported by almost all the browsers. `JavaScript` makes the web pages interactive, but it has to be used cautiously because browsers implement `JavaScript` differently even though the core `JavaScript` is same for all the browsers.

An important thing about `JavaScript` is that once you have learned its usage in browser programming, you can use it in other areas also. Microsoft's IIS server uses `JavaScript` to program server-side web pages (ASP); PDF files also use `JavaScript`; even Windows administration tasks can be automated by using `JavaScript`. A lot of applications, such as Dreamweaver and Photoshop, are scriptable with `JavaScript`. Some operating system add-ons on Linux and Windows even allow you to write small helper programs in `JavaScript`.

Nowadays, a lot of companies are also offering Application Programming Interfaces (APIs), which feature `JavaScript` objects and methods that you can use in your own pages. One of the examples of such API is

Google Maps. You can offer a zoomable and scrollable map in your application with just a few lines of code. Also JavaScript is easier to develop as compared to other high-level programming languages or server-side scripting languages, such as C++, Java, or to be run on server or command line, like Perl, PHP, etc.

Today there are so many web-oriented technologies available in the market that you should focus your client-side JavaScript efforts on tasks for which they are best suited. The following are a few situations when you should use JavaScript in your Web application:

- **Data entry validations**—A JavaScript can be used to validate the data entered in the form, before it is sent to the server for further processing. This saves a lot of time as well as effort of the server from extra processing.
- **HTML interactivity**—You can use DHTML for positioning the contents precisely on the web page, but if you want interactivity in your web page, you need JavaScript. By the word interactivity, we mean that the user is able to see the changes or get the feedback as soon as he/she enters the data in a form.
- **Serverless CGIs**—This term is used to describe the processes that would be programmed as CGIs (if JavaScript was not there) on the server-side, yielding slow performance because of the interactivity required between the program and the user. This includes the tasks, such as small data collection lookup, modification of images, and generation of HTML in other frames and windows-based on the user input. But, with the availability of JavaScript, all these tasks can now be done on the client-side.
- **CGI prototyping**—Sometimes you may want a CGI program to be at the root of the application because it reduces potential incompatibilities among browser brands and versions. It may be easier to create a prototype of CGI in client-side JavaScript. Use this opportunity to polish the user interface before implementing the application as a CGI.
- **Offloading a busy server**—If you have a highly trafficked web site, it may be beneficial to convert the frequently used CGI processes to the client-side JavaScripts. After a page is downloaded, the server is free to serve other visitors. Not only will this lighten the server load, but users also experience quicker response to the application embedded in the page.
- **Adding life to otherwise-dead pages**—HTML by itself is quite flat. Adding a blinking chunk of text doesn't help much; animated GIF images more often distract from, rather than contribute to the user experience at your site. But if you can dream up your ways to add some interactive zip to your page, it may engage the user and encourage recommendation to friends or repeat visits.

Next we'll discuss the merits of JavaScript. The following are the merits of JavaScript:

- **Less server interaction**—You can validate the user input before sending the page off to the server. This saves server traffic, which means saving money.
- **Immediate feedback to the visitors**—The user doesn't have to wait for a page reload to see if they have forgotten to enter something.
- **Automated fixing of minor errors**—For example, if you have a database system that expects a date in the format dd-mm-yyyy and the visitor enters it in the form dd/mm/yyyy, a clever JavaScript script could change this minor mistake prior to sending the form to the server. If that was the only mistake the visitor made, you can save her an error message—thus making it less frustrating to use the site.
- **Increased usability by allowing visitors to change and interact with the user interface without reloading the page**—A classic example of this would be select boxes that allow immediate filtering, such as only showing the available destinations for a certain airport, without making you reload the page and wait for the result.
- **Increased interactivity**—You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard. This is partly possible with CSS and HTML as well, but JavaScript offers you a lot wider—and more widely supported— range of options.
- **Richer interfaces**—You can use JavaScript to include items, such as drag-and-drop components and sliders—something that originally was only possible in thick client applications.
- **Lightweight environment**—Instead of downloading, a large file like a Java applet or a Flash movie, scripts are small in file size and get cached (held in memory) once they are loaded. JavaScript also uses the

browser controls for functionality rather than its own user interfaces, like Flash or Java applets do. This makes it easier for users, as they already know these controls and how to use them. In the next topic, we would discuss the fundamentals of the `JavaScript`.

Basics of JavaScript

`JavaScript` (ECMAScript) provides everything that you need in order to accomplish basic programming tasks, like data types, variables, operators that act on the variables, loops, functions that provide reusable code blocks to accomplish a task. Objects are also part of `JavaScript`. Before discussing all these elements, let's first understand the syntax of the `JavaScript`. The following are some essential points regarding the syntax of the `JavaScript`:

- `JavaScript` is case sensitive. Extra attention need to be paid when you are using capital letters in the name of variables, functions, and objects. For example, the variable named `XYZ` is different from the variable `xyz`.

In order to use `JavaScript` in your web document, you need to use the `<script>` tag. All the `JavaScript` coding is written between the opening and closing script tag. The following lines show the syntax of inserting `JavaScript` in your web document:

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

You can add the `JavaScript` code anywhere in the HTML document and browsers will interpret it. However, this is a bad practice of inserting `JavaScript` code in the document. You should always try to include the `JavaScript` code in the body of the HTML page.

- `//` is used to make the current line as the comment. The interpreter doesn't try to run the content written after the `//`. These comments are used to insert the notes in the document. These notes help in understanding the code and are also helpful for the other person who wants to read and understand the code.
- There is one more category of comments—the multi-line comments. The multi-line comments start with `/*` and end with `*/`. Such comments are useful when you don't want to execute a certain section of the code, but also not want to delete the code. For example, if you were having a problem with a block of code and you also do not know the cause of the problem, then in such a situation you can comment that block of the code using multi-line comments so as to isolate the problem.
- Curly braces ({ and}) are used to indicate a block of code. The code inside the braces is treated as one block. This will become clearer when we would discuss loops in the subsequent sections.
- Semicolons are optional in `JavaScript`, but it is a good practice to use semicolons in the document because this makes the code easier to read and debug. A semicolon is used to define the end of a statement. Although you can put many statements in a single line, it is better to put each statement on a separate line.

Now based on the preceding syntax, we'll give you the first `JavaScript` example. The code for this example is as follows:

```
<html>
<body>
<script type="text/javascript">
document.write("Hello World!")
</script>
</body>
</html>
```

This is the simplest example of `JavaScript`. The preceding code will produce an output 'Hello World!'. The tags `<script type="text/javascript">` and `</script>` shows the starting and end point of the

JavaScript code. The command, document.write, is the standard JavaScript command for writing the output to a page. Now, we'll discuss the programming elements available in JavaScript language one by one.

Data Types

JavaScript provides the following data types for handling data. The data is stored in variables and the data type defines the kind of data which a variable can store:

- Number—The Number type includes both integers and floating-point numbers.
- String—A String type includes any group of one or more characters. A String type data is always shown by enclosing it in quotation marks, e.g. "8" is a String type, while 8 is of Number type.
- Boolean—A Boolean data type has only two values—true and false.
- Undefined—The Undefined type has only one value, i.e. Undefined. The Undefined value occurs only when a variable is declared, but has not been assigned any value.
- Null—The Null type also has one value, i.e. null. A null value means that the object in the question doesn't exist.

In addition to these data types, two more data types are also available in JavaScript—Object and Function type. These data types are sometimes called Reference data types. The Reference data types contain the reference to the memory location that holds the particular data.

You don't need to declare a data type, while declaring or initializing a variable in JavaScript. JavaScript uses dynamic typing, i.e. the data type is inferred by the context of the JavaScript statement. This facilitates the use of the same variable for different data types and at different times, as shown here:

```
var num = "lucy";
// num contains a string
num = 9;
// num contains a number
```

Variables

A variable can be referred as the container for storing information. This information can be of any type available in JavaScript. The value of a variable can change during the script. The lifetime of a variable depends on a number of factors. But the instant a web page clears the window, any variable it knows about are discarded. You have a couple of ways for creating a variable in JavaScript. Use the var keyword, followed by the name you want to give to the variable. For example, you can declare a variable given in the following statement:

```
var myVar;
```

The preceding statement lets the browser know that you can use the myVar variable later for holding the information or to modify any data stored in that variable.

In order to assign a value to any variable, use the assignment operators. For example, if you want to assign a value to the myVar variable at the time of declaration of the variable, then the statement would be as follows:

```
var myVar = 99;
```

Another way to create a variable is to first declare the variable and then assign the value to the variable by using the assignment operators as shown here:

```
var myVar;
myVar = 99;
```

The preceding code shows that you need to use the var keyword only once, for the whole life of the variable in the document.

The variable names are case sensitive in JavaScript. All the variable names must begin with a letter or underscore. Camel notation is generally used for variable names, although no specific notation is required. In camel notation, the first word is lowercase, and the additional words start with a capital letter, as shown here:

```
var myFavouriteColor;
```

Operators

Operators enable you to perform an action on a variable. The basic JavaScript operators, include assignment, arithmetic, comparison, logical, and increment/decrement operators. Let's discuss these operators one by one:

- **Assignment operators** – The assignment operator is used to assign a value to a variable as shown here:

```
var myFavouriteColor ="Red";
```

- **Arithmetic operators** – The following arithmetic operators are available in `JavaScript` for performing the basic arithmetic function:
  - Addition (+)
  - Subtraction (-)
  - Multiplication (*)
  - Division (/)

The example given here shows the use of all the arithmetic operators:

```
var x = 5;
var y = 10;
var z = x + y;
//z = 15
x = "My";
y = "name";
z = x + y;
// z ="My name"
```

NOTE

The Addition operator and the string concatenation operator are the same character (+). If you use this operator with numeric values, the numbers are added together. If you use this operator with string values, the strings are joined together (concatenated) into a single string.

`JavaScript` includes a Modulus (`%`) operator, also known as the Remainder operator. The Modulus operator is used to calculate the value of the remainder after a number is divided by another number, as shown here:

```
var myMod = 11%2;
//myMod = 1
```

- **Comparison operator** – Comparison operators are used to evaluate an expression and return a Boolean value (true or false) indicating whether the comparison is true or false. For example, if `a = 2` and `b = 4`, then the expression `a > b` is false. The comparison operator, in this case, is the greater than (>) operator. Table 29.1 lists the operators available in `JavaScript`:

Table 29.1: JavaScript operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

- **Logical operators** – In addition to the comparison operators, JavaScript also includes logical operators for more complex comparisons. The logical operators include && (AND), || (OR), and ! (NOT). These operators also return a Boolean value (true or false). The AND operator returns true only if all the operands are true. The OR operator returns true if any of the operands is true. The `NOT` operator always returns the value opposite to that of the operand.
- **Increment and decrement operators** – The increment and decrement operators provide a shortcut for adding or subtracting 1 from a value, as shown here:

```
var a = 1;
a++;        // a = 2
```

```
var b = 45;
b--;        // b = 44
```

Increment and decrement operators can appear before a variable (prefix) or after a variable (postfix). If a variable with a prefix operator is used in an expression, the value of the variable is incremented or decremented before the expression is evaluated. If a postfix operator is used, the value is incremented or decremented after the expression is evaluated.

Conditional Statements and Loops

A `JavaScript` program is composed of statements. Variable declarations, assignments, and initializations are examples of `JavaScript` statements. `JavaScript` also includes a core set of programming statements similar to those used in other programming languages—conditionals (`if/else`, `switch`), and loops (`for`, `while`). First we'll discuss the conditional statements and their use in `JavaScript`.

- `if` **statement**—An `if` statement evaluates the Boolean value of an expression (true or false), and then executes the code based on the result of that evaluation. If the expression in the parentheses evaluates to true, the block of code contained in the curly braces ({ }) following the expression is executed. The syntax of `if` statement is given here:

```
if ( condition to be checked) {
  Code to be executed if the condition is true
}
```

Now, we'll show you how to use the if statement inside the script tag. For the sake of simplicity, we are not giving the full structure of the page. The example will write the greeting 'Good morning' if the time is less than 12. The code for this example is as follows:

```
<script type="text/javascript">
//Write a "Good morning" greeting if
//the time is less than 12
var d=new Date()
var time=d.getHours()
if (time<12)
{
  document.write("<b>Good morning</b>")
}
</script>
```

In the preceding code, a new instance of `date` object is created and is assigned to the variable `d`. The objects used in the `JavaScript` would be discussed in the subsequent topics of this chapter. The `Date()` object handles the date and time. The variable time is used to get the time in hours. The condition in `if` statement checks whether the time is less than 12. If the condition is true, then the greeting 'Good morning' would be displayed.

- `if...else` **statement**—The if-else statement is used if you want to execute some code when the condition is true and the other code if the condition is false. The syntax of the if-else statement is as follows:

```
if(condition)
{
  Code to be executed if the condition is true
}
Else
{
  Code to be executed if the condition is not true
}
```

The following code shows how to use `if-else` statement in the `JavaScript`. This is an extension of the example that we have discussed for `if` statement. An additional `else` statement is added to the previous example only:

```
<script type="text/javascript">
//If the time is less than 12,
//you will get a "Good morning" greeting.
//Otherwise you will get a "Good day" greeting.
var d = new Date()
var time = d.getHours()
if (time < 12)
{
  document.write("Good morning!")
}

else
{
  document.write("Good day!")
}
</script>
```

This code is the same as that for `if` statement, except that if the condition is false, i.e. if the time is not less than 12, it will display 'Good day!' greeting.

- `Switch` **statement**—A switch statement can be used in place of a series of `if` statements. The syntax of the `switch` statement is as follows:

```
switch(n) {
case 1:
  execute code block 1
  break
case 2:
  execute code block 2
  break
default:
  code to be executed if n is
  different from case 1 and 2
}
```

First we have a single expression `n` (most often a variable), that is evaluated once. The value of the expression is then compared with the values for each case in the structure. If there is a match, the block of code associated with that case is executed. Use break to prevent the code from running into the next case automatically. The default case plays a similar role to an `else` statement. If there is not a specific case for a value, the default is used.

Now, we'll discuss the loops available in `JavaScript`. Loops are a way to repeat a block of code based on the conditions that you specify. The two major kinds of loops are—for and while. Let's discuss the `for` loop first.

- `for` **loop**—A `for` loop is used when you know in advance how many times the script should run. A `for` loop includes three components, like initialization, test condition, and iterator. The syntax of a `for` loop is as follows:

```
for (var=startvalue;var<=endvalue;var=var+increment) {
    code to be executed
}
```

The example given here shows the use of a `for` loop in `JavaScript`. It defines a loop that starts with `i=0`. The loop will continue to run as long as `i` is less than, or equal to `10`. Here `i` will increase by 1 each time the loop runs. See the following code snippet:

```
<html>
<body>
<script type="text/javascript">
var i=0
for (i=0;i<=10;i++) {
  document.write("The number is " + i)
  document.write("<br />") }
</script>
</body>
</html>
```

The code in this loop is executed 11 times and each time the current value of `i` is written to the page.

- `while` **loop**—A while loop is used when you want the loop to execute and continue executing, while the specified condition is true. The syntax of a `while` loop is as follows:

```
while (var<=endvalue) {
    code to be executed
  var++;
}
```

The following example shows the usage of a `while` loop in `JavaScript`. It defines a loop that starts with `i=0`. The loop will continue to run as long as `i` is less than, or equal to `10`. Here `i` will increase by 1 each time the loop runs:

```
<html>
<body>
<script type="text/javascript">
var i=0
while (i<=10)
{
  document.write("The number is " + i)
  document.write("<br />")
  i=i+1
}
</script>
</body>
</html>
```

In this case, the initialization is `i=0`. The initialization occurs before the start of the `while` loop. The test condition is `i<=10` and the iterator is `i+1`.

The output of the preceding code will be same as the output of the `for` loop earlier.

NOTE

Be careful not to create an infinite while loop. Check to make sure that your test condition is not always true.

- `do-while` **loop**—The do...while loop is a variant of the `while` loop. This loop will always execute a block of code once, and then it will repeat the loop as long as the specified condition is true. This loop will always be executed at least once, even if the condition is false, because the code is executed before the condition is tested. The syntax of the do-while loop is as follows:

```
do
{
   code to be executed
  var++;
}
while (var<=endvalue)
```

Now, we'll give you an example showing the use of a do-while loop. The following is the code for this example:

```
<html>
<body>
<script type="text/javascript">
var i=0
do
{
  document.write("The number is " + i)
  document.write("<br />")
  i=i+1
}
while (i<0)
</script>
</body>
</html>
```

The earlier code is understandable and doesn't need further explanation. You can see that the condition is checked at the end of the loop, which makes sure that the loop will run at least once.

Functions

A function is a reusable group of code statements that are treated as a unit. `JavaScript` includes many built-in functions, and you can also create your own functions for code blocks that you want to reuse. Function definitions are usually included in script blocks in the head section of an HTML/XHTML page. A function must be defined before it can be used. Because the code in the head section of the page is read and interpreted before the code in the body section, functions are usually defined in the head section and called from the body section.

The syntax for creating a function is as follows:

```
function functionname(var1,var2,...,varX)
{
  some code
}
```

Here, `var1`, `var2`, etc. are the variables or values passed into the function. The curly brackets `{ and the }` defines the start and end of the function.

The following example shows how a function is created and defined in `JavaScript`. This example also shows how a function is called in the subsequent portion of the program:

```
<html>
<head>
<script type="text/javascript">
function displaymessage() {
  alert("Hello World!")
}
</script>
</head>
<body>
<form>
<input type="button" value="Click me!"onclick="displaymessage()" >
</form>
</body>
</html>
```

In the earlier example, a function named `displaymessage()` was defined in the head section of the page. This function shows an alert message only. The `displaymessage()` function is called in the body section of the page. This function is called on the `onclick` event of the button, i.e. the message will be displayed only when the user clicks on the 'Click me!' button.

You can also make a function that will return some value. Hence, the function that will return some value should use the return statement. The function `incomeTax` has a single parameter `income`. This function calculates the income tax and returns the value of `totalTax`.

Here's the code snippet that shows a function with `return` statement:

```
function incomeTax (income) {
  var incTax = income * .2;
  var surcharge = incTax * .02;
  var totalTax = incTax + surcharge;
  return totalTax;
}
```

JavaScript Objects

An object is a collection of properties and methods that are grouped together with a single name. The objects available in the `JavaScript` can be divided into these three categories:

- Built-in objects
- Browser objects
- User-defined objects

Now, we'll explain the preceding `JavaScript` objects one by one.

Built-in Objects

There are nine built-in objects available in `JavaScript`. Built-in objects are those objects, which are available regardless of the content of window. These objects operate independently of whichever pages your browser has loaded. These objects are also called core language objects. Some of the commonly used objects are Array, String, Math, and Date. Let's understand it better.

- `Array`—When you use the `new` keyword and an object constructor function, you create a new instance of the object:

```
var myArray = new Array();
// creating a new instance of the Array object
```

In this case, the instance is a specific Array object that is assigned to a variable named `myArray`. This object has all the methods and properties of all Array objects. Whatever changes you make to this instance of the Array object affect only this instance, other instances of Array object remains unchanged. The changes don't affect the Array object itself. Methods are the specific functions that belong to a particular type of object. The Array object, for example, includes a `reverse()` function that reverses the order of members of the array:

```
var backwards = myArray.reverse();
```

Objects also include properties. Properties are attributes of an object, e.g. the Array object includes a `length` property:

```
var howLong = myArray.length;
```

You can use dot notation to access either the methods or properties of an object:

```
myArray.reverse();
myArray.length;
```

- `String`—The String object is used to manipulate a stored piece of text. The following example uses the `length` property of the String object to find the length of a string:

```
var txt="Hello world!"
document.write(txt.length)
```

The output is as follows:

```
12
```

The following example uses the `toUpperCase()` method of the String object to convert a string into uppercase letters:

```
var txt="Hello world!"
document.write(txt.toUpperCase())
```

The output is as follows:

```
HELLO WORLD!
```

- `Math`—The Math object includes several mathematical values and functions. This object allows you to perform common mathematical tasks. You do not need to define the Math object before using it. `JavaScript` provides eight mathematical values (constants) that can be accessed from the Math object. These are E, PI, square root of 2, square root of 1/2, natural log of 2, natural log of 10, base-2 log of E, and base-10 log of E. You may reference these values from your `JavaScript` like this:

```
Math.E
Math.PI
Math.SQRT2
Math.SQRT1_2
Math.LN2
Math.LN10
Math.LOG2E
Math.LOG10E
```

In addition to the mathematical values that can be accessed from the Math object, several functions (methods) are also available. The following example uses the `round()` method of the Math object to round a number to the nearest integer:

```
document.write(Math.round(4.7))
```

The output is as follows:

```
5
```

The following example uses the `random ()` method of the Math object to return a random number between 0 and 1:

```
document.write(Math.random())
```

The output is as follows:

```
0.4906046217190822
```

- `Date`—The `Date` object is used to work with dates and times. We define a `Date` object with the `new` keyword. The following code line defines a `Date` object called `myDate`:

```
var myDate=new Date()
```

We can easily manipulate the date by using the methods available for the `Date` object. In the example here, we set a `Date` object to a specific date (29th January 2020):

```
var myDate=new Date()
myDate.setFullYear(2020,0,29)
```

The `Date` object is also used to compare two dates. The following example compares today's date with 20th January 2015:

```
var myDate=new Date()
myDate.setFullYear(2015,0,20)
var today = new Date()
if (myDate>today)
  alert("Today is before 20th January 2015")
else
  alert("Today is after 20th January 2015")
```

Browser Objects

The Browser Object Model (BOM) is a collection of objects that interact with the browser window. These objects include the window object, history object, location object, navigator object, screen object, and document object. The window object is the top object in the BOM hierarchy. The window object can be used to move and resize windows as well as create new windows. The window object also includes methods to create dialog boxes, such as the alert dialogs. The window object is a unique object—you don't have to explicitly include a reference to it when you use its methods or properties. Consider the following example:

```
alert ("Look! Is there a reference?");
```

This function is exactly the same as the following:

```
window.alert("Look! Is there a reference?");
```

The history object keeps track of every page the user visits. Methods of this object include forward, back, and go:

```
history.back(2);
// go back by 2 pages
```

The location object contains the URL of the page. You can use its `href` property to go to a new page:

```
location.href="myPage.htm";
```

The navigator object contains information about the browser name and version. Properties include appName, appVersion, and userAgent. The screen object provides information about the display characteristics, such as screen height and width in pixels. The document object is included in the window object, as shown in Figure 29.1:

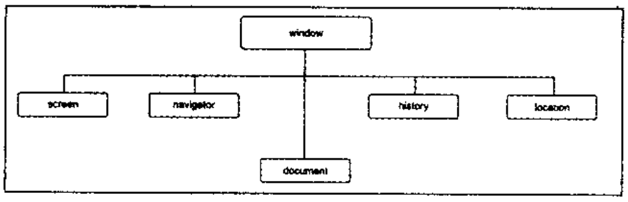


Figure 29.1: Various Browser Objects

Let's discuss the objects shown in Figure 29.1 in detail.

- **The window object**—At the top of the object hierarchy is the window object. This object is very important as it plays the role of master container for all the contents you view in the Web browser. In addition to the content part of the window, where the documents will go, a window's sphere of influence includes the dimension of the window and all the stuff that surrounds the content area. The area where the scrollbars, tool bars, status bar, and menu bar are present is known as the window's chrome.

The window object is a convenient place for the DOM to attach the methods that will display model dialog boxes and adjust the text that displays at the bottom of the browser window. A window object method enables you to create a separate window that appears on screen.

You can reference the properties and methods of the window object in many ways—depending more on the whim and style as compared to the syntactical requirements. The most common way to compose such references include the window object in the reference as shown here:

```
window.propertyName
window.methodName(parameters)
```

A window object also has a synonym when the script doing the referencing points to the window that houses the document. The synonym is `self`. In this case, you replace the window object by its synonym `self` and the syntax changes to what is shown here:

```
self.propertyName
self.methodName([parameters])
```

A user doesn't create the main browser window. The user does so by opening the browser or by opening a URL or file from the browser's menus. The script can generate any number of subwindows when the main window is already open. The method that generates the new window is `window.open()`. This method can contain up to three parameters, which will define the characteristics, such as the URL of the document to load, its name for the target attribute reference purposes in HTML tags, and the physical appearance. The following code shows the creation of a new sub window:

```
var subWindow = window.open("definition.html", "def", "height=225, width=400");
```

This statement will open a new sub window. You can also close this window by using the method `close()` as follows:

```
subWindow.close();
```

Some of the important methods available with the window object are `alert()`, `confirm()`, and `prompt()`. The `alert()` method generates a dialog box that displays the text that is passed as a parameter. A single OK button enables the user to dismiss the alert.

The next method is `confirm()`. This dialog box represents two buttons—'Cancel' and 'OK' and is called a confirm dialog box. This is one of the methods that return a value: true if the user clicks on 'OK' and false when the user clicks on the 'Cancel' button.

The `prompt()` dialog box displays the message that you set and provides a text field for entering the user's response. The two buttons 'OK' and 'Cancel' enable the user to dismiss the dialog box. The user either cancels the entire operation by clicking on "Cancel" button or accepts the input typed in the dialog box by clicking on the "Ok" button.

Some of the properties available with the window object are as follows:

- `closed`—It returns a Boolean value that indicates whether or not a window is closed.
- `defaultStatus`—It sets or returns the default text in the status bar of the window.
- `document`—It is used to access document object.
- `history`—It is used to access history object.
- `location`—It is used to access location object.
- `name`—It sets or returns the name of the window.
- `opener`—It returns a reference to the window that created the window.
- `outerheight`—It sets or returns the outer height of a window.
- `outerwidth`—It sets or returns the outer width of a window.

- `self`—It returns a reference to the current window.
- `status`—It sets the text in the status bar of a window.

Some commonly used methods of the window object are as follows:

- `alert()`—The `alert()` method is used to display an alert box with a specified message and an OK button. The syntax of the `alert()` method is as follows:

```
alert(message)
```

- `close()`—The `close()` method is used to close the current window. The syntax of the `close()` method is as follows:

```
window.close()
```

- `confirm()`—This method displays a dialog box with a message and an 'OK' and a 'Cancel' button. The syntax of the `confirm()` method is as follows:

```
confirm(message)
```

- `open()`—This method opens a new browser window. The syntax of the `open()` method is as follows:

```
window.open(URL, name, specs, replace)
```

All the four parameters are optional.

- `print()`—The `print()` method is used to print the contents of the current window. The syntax of the `print()` method is as follows:

```
window.print()
```

- `prompt()`—This method displays a dialog box that prompts the user for input. The syntax of the `prompt()` method is as follows:

```
prompt(text, defaultText)
```

Both the parameters passed are optional.

- **The location object**—The location object represents the URL loaded into the window. A URL consists of many components that define the address and method of the data transfer for a file. A URL can be divided into two parts—protocol and hostname. You can access all these items as the properties of the location object. However, in most of the cases, your script would be interested in only one property, the `href` property, which defines the complete URL.

Setting the `location.href` property is the most common way for your script which navigates to other pages:

```
location.href = http://www.kogentindia.com;
```

You can generally navigate to other pages in your own web site by specifying the relative URL rather than the complete URL. For the pages outside the domain, you need to specify the complete URL. If the page is loaded in another window or frame, the window reference should be part of the statement. For example, if your script opens a new window and assigns its reference to a variable named new Window, the statement that loads a page into the sub window is as follows:

```
newWindow.location.href = http://www.kogentindia.com;
```

Some of the important properties available with the location object are as follows:

- `host`—This property sets or returns the hostname and port number of the current URL.
- `hostname`—This property sets or returns the hostname of the current URL.
- `href`—It sets or returns the entire URL.
- `pathname`—It sets or returns the path of the current URL.
- `protocol`—It sets or returns the protocol of the current URL.

Some of the important methods available with the location object are `assign ()`, `reload ()`, and `replace ()`. The `assign ()` method loads a new document. The syntax of the assign is as follows:

```
location.assign(URL)
```

The parameter URL represents the URL for the document to be loaded.

The `reload()` method is used to reload the current document. The syntax for `reload()` method is as follows:

```
location.reload()
```

The `replace()` method replaces the current document with the new one. The syntax for the `replace()` method is as follows:

```
location.replace(newURL)
```

The next object to be discussed is the navigator object.

- **The navigator object**—The navigator object is implemented in almost every scriptable browser, even though its name is reminiscent of the Netscape Navigator-branded browser. All browsers also implement a handful of properties which reveal the same kind of properties that the browser sends to the servers, with each page request. Thus, the `navigator.userAgent` property returns a string with a number of details about the browser and the operating system. For example, the script running in Internet Explorer 7 in Windows XP receives the following value for the `navigator.userAgent` property:

```
Mozilla/4.0 (compatible;
MSIE 7.0;
Windows NT 5.2;
.NET CLR 1.1.4322;
.NET CLR 2.0.50727;
.NET CLR 3.0.04506.590;
.NET CLR 3.5.20706)
```

- **The document object**—The document object holds the real contents of the page. Properties and methods of the document generally affect the look and content of the document that occupies the window. All W3C DOM-compatible browsers allow script access to the text contents of a page when the document is loaded. As you have seen in the preceding code snippets that the `document.write()` method lets a script create content dynamically as the page loads on the browser. Many document object properties are the arrays of other objects in the document. These properties provide additional ways to reference these objects.

You can access the document object's properties and methods straightforward, as shown in the following syntax examples:

```
window.document.propertyName
window.document.methodName(parameters)
```

There are four important collections available with the document object. These collections are as follows:

- `anchors[ ]`—The anchors collection returns a reference to all Anchor objects in the document. The syntax anchor[] collection is as follows:

```
document.anchors[]
```

- `forms[ ]`—The forms collection returns a reference to all Form objects in the document. The syntax for the `form[ ]` is as follows:

```
document.forms[]
```

- `images[ ]`—The images collection returns a reference to all Image objects in the document. The syntax for the `images[ ]` is as follows:

```
document.images[]
```

- `links[ ]`—The links collection returns a reference to all Area and Link objects in the document. The syntax for the `links[ ]` is as follows:

```
document.links[]
```

In addition to these collections, document object also has some properties. These are as follows:

- `cookie`—The `cookie` property sets or returns all cookies associated with the current document. The syntax for using the `cookie` is as follows:

```
document.cookie
```

- `domain`—The `domain` property returns the domain name for the current document. The syntax of using the `domain` property is as follows:

```
document.domain
```

- `lastModified`—The `lastModified` property returns the date and time the document was last modified. The syntax of `lastModified` is as follows:

```
document.lastModified
```

  - referrer—The referrer property returns the URL of the document that loaded the current document. The syntax for using the referrer property is as follows:

```
document.referrer
```

  - title—The title property returns the title of the current document (the text inside the HTML title element). The syntax of using the title property is as follows:

```
document.title
```

  - URL—The URL property returns the URL of the current document. The syntax of using the URL property is as follows:

```
document.URL
```

- The important methods available with the document object are as follows:
  - close() —The close() method closes an output stream opened with the document.open method, and displays the collected data. The syntax of the close() method is as follows:

```
document.close()
```

  - open() —The open() method opens a stream to collect the output from any document.write or document.writeln methods. The syntax of writing the open() method is as follows:

```
document.open(mimetype, replace)
```

  - write() —The write() method writes HTML expressions or JavaScript code to a document. The syntax of the write() method is as follows:

```
document.write(exp1,exp2,exp3,....)
```

  - writeln() —The writeln() method is identical to the write() method, with the addition of writing a new line character after each expression. The syntax of writeln() method is as follows:

```
document.writeln(exp1,exp2,exp3,....)
```

- **The history object**—As the user surfs the web, the browser maintains a list of URLs for the most recent stops. This list is represented in the scriptable object model by the history object. A script cannot surreptitiously extract actual URLs maintained in that list unless you use signed scripts and the user grants permission. Under unsigned conditions, a script can methodically navigate to each URL in the history, in which case the user sees the browser navigating on its own, as though possessed by a script.
- You should use the history object and its methods with extreme care. The design should be smart enough to watch what the user is doing with your pages. Otherwise, you run the risk of confusing your user by navigating to unexpected places. Your script can also get into trouble because it cannot detect where the current document is in the Back-Forward sequence in history.
- The history object is part of the window object and is accessed through the window.history property. The following important methods are associated with the history object:
  - back() —The back() method loads the previous URL in the history list. The syntax for using the back() method is as follows:

```
history.back()
```

  - forward() —The forward() method loads the next URL in the history list. The syntax for using the forward() method is as follows:

```
history.forward()
```

  - go() —The go() method loads a specific page in the history list. The syntax for using the go() is as follows:

```
history.go(number|URL)
```

- **The screen object**—This is another read-only object that lets the script learn about the physical environment in which the browser is running. For example, this object reveals the number of pixels high and wide available in the monitor. The important properties available with the screen object are as follows:
  - availHeight—The availHeight property returns the height of the client's display screen excluding the Windows Taskbar. The syntax of using this property is as follows:

```
screen.availHeight
```

  - availWidth—The availWidth property returns the width of the client's display screen excluding the Windows Taskbar. The syntax of using this property is as follows:

```
screen.availWidth
```

- bufferDepth—The bufferDepth property sets or returns the bit depth of the color palette in the off-screen bitmap buffer. The syntax for this property is as follows:

```
screen.bufferDepth=number
```

- colorDepth—The colorDepth property returns the bit depth of the color palette on the destination device or buffer. The syntax for this property is as follows:

```
screen.colorDepth
```

- height—The height property returns the height of the client's display screen. The syntax for height property is as follows:

```
screen.height
```

- width—The width property returns the width of the client's display screen. The syntax for using this property is as follows:

```
screen.width
```

Now, a JavaScript uses the Document Object Model (DOM) for accessing objects associated with any HTML page and changes their properties. We will now discuss Document Object Model in detail.

What is Document Object Model?

Document Object Model (DOM) is a platform- and language-independent standard object model for representing HTML or XML in tree formats. Since the DOM supports navigation in any direction (e.g. parent and previous sibling) and allows the arbitrary modifications, an implementation must at least buffer the document that is read so far (or some parsed form of it). Hence, the DOM is likely to be best suited for applications where the document must be accessed repeatedly or out of sequence order. If the application is strictly sequential and one-pass, the SAX model is likely to be faster and use less memory.

W3C began the development of the DOM in the mid-1990s. Although W3C never produced a specification for DOM 0, it was nonetheless a partially documented model and was included in the specification of HTML 4. By October 1998, the first specification of DOM (DOM 1) was released. DOM 2 was issued in November 2000, with specifications on the style sheet object model and style information manipulation. DOM 3 was released in April 2004 and is the current release of the DOM specification.

The Document Object Model (DOM) is a tree-based representation of a document. The DOM was created by the World Wide Web Consortium (W3C) for XML and HTML/XHTML. The DOM provides a set of objects for representing the structure of the document, as well as for accessing those objects. The DOM is divided into the following three parts:

- The Core DOM, which includes objects that XML and HTML have in common.
- The XML DOM includes the XML objects.
- The HTML DOM includes the HTML objects.

  All the elements in a page are related to the topmost object. You can access any object in an HTML page and change its properties by using JavaScript in this model. For example, you can:

  - Change its position
  - Change its source file
  - Change its style properties
  - Change its content
  - Add new content

A document can be viewed as a node tree. In the node tree view, a document is a collection of nodes. The nodes symbolize the branches and leaves on the document tree. There are several types of nodes, but the three main types are—Element nodes, Text nodes, and Attribute nodes. Figure 29.2 shows the document as the node tree:

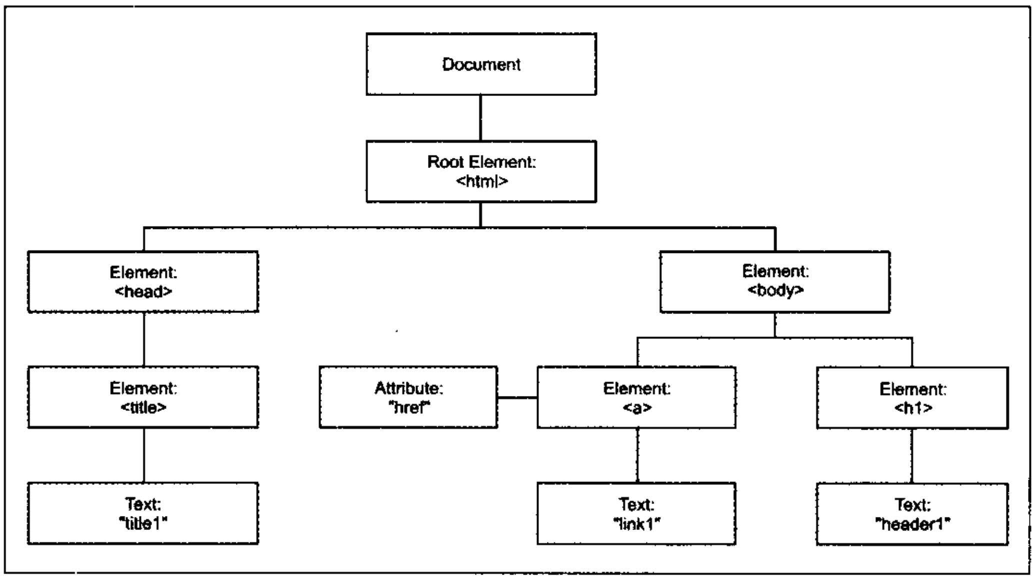


Figure 29.2: A Node Tree for an HTML Document

From Figure 29.2, the following are the key components of the node structure:

- **Element nodes**—Elements are the basic building blocks of documents and give them their structure. Elements can contain other elements, e.g. '<html>', '<head>', '<body>', etc. are the Element nodes.
- **Text node**—In HTML/XHTML, Text nodes are always contained in Element nodes, e.g. 'title1', 'link1', and 'header1' are all Text nodes.
- **Attribute nodes**—Attributes provide more information about elements. Attribute nodes are always contained in Element nodes. For example, the 'href' is an Attribute node.

Every node has some properties that contain some information about the node. The properties are as follows:

- nodeName
- nodeValue
- nodeType

Let's now understand these properties.

- `nodeName`—The `nodeName` property contains the name of a node.
  - The `nodeName` of an Element node is the tag name.
  - The `nodeName` of an Attribute node is the attribute name.
  - The name of a Text node is always `#text`.
  - The name of the Document node is always `#document`.
- `nodeValue`—On Text nodes, the `nodeValue` property contains the text. On Attribute nodes, the `nodeValue` property contains the attribute value. The `nodeValue` property is not available on Document and Element nodes.
- `nodeType`—The `nodeType` property returns the type of node. The most important node types are described in the Table 29.2:

Table 29.2: Nodes of HTML document

[illegible]	Node Type
Element	1
Attribute	2

Table 29.2: Nodes of HTML document

[illegible]	[illegible]
Text	3
Comment	8
Document	9

Methods Available in DOM for Accessing Objects

There are two methods available in DOM for accessing various elements of the document. These two methods are getElementById and getElementsByTagName. Let's discuss these methods.

getElementById() Method

The getElementById() method returns the element with the specified ID. The syntax of this method is as follows:

```
document.getElementById("someID");
```

If an element has an id, the simplest way to access it is by using the getElementById() method, as shown here:

```
<p id="someID">Greetings, Hello!</p>
...
var a = document.getElementById('someID');
```

Here, a is now a shortcut for accessing the unique element with the id value of someID. In order to change a property of this element, for example the fontWeight, use the following code snippet:

```
x.style.fontWeight = "bold";
// changes the font weight to bold
```

You can also use getElementById to access elements by using node properties. For example, if the head and body elements in Figure 29.2 include id values, such as the following, you can use these id values to access all the other elements in the page:

```
<head id="e1">
...
<body id="e2">
```

To access the parent node of the head and body elements, you can use either of the following:

```
document.getElementById('e1').parentNode;
document.getElementById('e2').parentNode;
```

Both of these access the html element. To access all the children of the body element, use the following:

```
document.getElementById('e2').childNodes;
```

The child nodes are contained in an array. You can access individual child nodes by using the array index value:

```
document.getElementById('e2').childNodes[0];
```

This accesses the first child node of the body element <a>, since array indexes start with 0. If you want to access a sibling node, use the following:

```
document.getElementById('e2').previousSibling;
// accesses the head element
document.getElementById('e1').nextSibling;
// accesses the body element
```

Next, we'll study the getElementsByTagName() method.

getElementsByTagName() Method

The getElementsByTagName() method returns all elements (as a nodeList) with the specified tag name that are descendants of the element when you are using this method. The getElementsByTagName() can be used on any HTML element, and also on the document. The syntax of getElementsByTagName() method is as follows:

```
document.getElementsByTagName("tagname");
or:
document.getElementById('someID').getElementsByTagName("tagname");
```

To use the `getElementsByTagName` method, you use a `tagname` rather than an `id`. For example:

```
var y = document.getElementsByTagName('h1');
```

Here, y contains a reference to `h1` element in the document. To change the text color of `h1`, use the following:

```
y.style.color = "green";
```

Now, we'll discuss the various DOM levels.

DOM Levels

The W3C DOM specifications are divided into levels, each of which contains the required and optional modules. To claim and support a level, an application must implement all the requirements of the claimed level and the levels below it. An application may also support vendor-specific extensions, which don't conflict with the W3C standards. As of 2005, Level 1, Level 2, and some modules of Level 3 are W3C recommendations, which mean they have reached their final form. The following are the levels of DOM:

- **Level 0**—The application supports an intermediate DOM, which existed before the creation of DOM Level 1. Examples include the DHTML Object Model or the Netscape intermediate DOM. Level 0 is not a formal specification published by the W3C, but rather a shorthand that refers to what existed before the standardization process.
- **Level 1**—It includes the Navigation of DOM (HTML and XML) document (tree structure) and content manipulation (includes adding elements). HTML-specific elements are included as well.
- **Level 2**—XML namespace support, filtered views and events.
- **Level 3**—This level consists of 6 different specifications:
  - DOM Level 3 Core
  - DOM Level 3 Load and Save
  - DOM Level 3 XPath
  - DOM Level 3 Views and Formatting
  - DOM Level 3 Requirements
  - DOM Level 3 Validation, which further enhances the DOM

These levels will be discussed in detail in the following section.

DOM Level 1

The DOM Level 1 is an application programming interface that allows programs and scripts to dynamically access and update the content, structure and style of HTML and XML 1.0 documents. The Document Object Model provides a standard set of objects for representing HTML and XML documents, a standard model of how these objects can be combined, and a standard interface for accessing and manipulating them. Vendors can support DOM as an interface to their proprietary data structures and APIs, and content authors can write to the standard DOM interfaces rather than product-specific APIs, thus increasing interoperability on the Web.

The goal of the DOM specification is to define a programmatic interface for XML and HTML. The DOM Level 1 specification is separated into two parts—Core and HTML. The Core DOM Level 1 section provides a low-level set of fundamental interfaces that can represent any structured document, as well as define extended interfaces for representing an XML document. These extended XML interfaces need not be implemented by a DOM implementation that only provides access to HTML documents; all of the fundamental interfaces in the Core section must be implemented. A compliant DOM implementation that implements the extended XML interfaces is required to also implement the fundamental Core interfaces, but not the HTML interfaces. The HTML Level 1 section provides additional, higher-level interfaces that are used with the fundamental interfaces defined in the Core Level 1 section to provide a more convenient view of an HTML document. A compliant implementation of the HTML DOM implements all of the fundamental Core interfaces as well as the HTML interfaces.

DOM Level 2

The DOM Level 2 extends Level 1 with support for XML 1.0 with namespaces and adds support for Cascading Style Sheets (CSS), events (user interface events and tree manipulation events), and enhances tree manipulations (tree ranges and traversal mechanisms). The DOM Level 2 defines the following specifications, which have reached in their final form:

- DOM Level 2 Core
- DOM Level 2 Views
- DOM Level 2 Events
- DOM Level 2 Style
- DOM Level 2 Traversal and Range
- DOM Level 2 HTML

DOM Level 2 Core

This specification defines the Document Object Model Level 2 Core, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content and structure of documents. The Document Object Model Level 2 Core extends the Document Object Model Level 1 Core.

The DOM Level 2 Core is made of a set of core interfaces to create and manipulate the structure and contents of a document. The Core also contains specialized interfaces dedicated to XML.

DOM Level 2 Views

This specification defines the Document Object Model Level 2 Views. This is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content of a HTML and XML document. The DOM Level 2 Views extends the DOM Level 2 Core.

A document may have one or more *views* associated with it, e.g. a computed view on a document after applying a CSS stylesheet, or multiple presentations (e.g. HTML Frame) of the same document in a client. That is, a view is some alternate representation of a source document.

A view may be static, reflecting the state of the document when the view was created, or dynamic, reflecting changes in the target document as they occur, subsequent to the view being created. This level of the DOM specification makes no statement about these behaviors.

This section defines an `AbstractView` interface, which provides a base interface from which all such views shall derive. It defines an attribute, which references the target document of the `AbstractView`. The only semantics of the `AbstractView` defined here create an association between a view and its target document. There is no sub interfaces of `AbstractView` defined in the DOM Level 2.

However, `AbstractView` is defined and used in this Level in two places:

- A document may implement a `DocumentView` that has a default view attribute associated with it. This default view is typically dependent on the implementation, e.g. the browser frame rendering the document. The default view can be used in order to identify and/or associate a view with its target document (by testing object equality on the `AbstractView` or obtaining the `DocumentView` attribute).
- A `UIEvent` typically occurs upon a view of a Document, e.g. a mouse click on a browser frame rendering a particular `Document` instance. A `UIEvent` has an `AbstractView` associated with it which identifies both the particular (implementation-dependent) view in which the event occurs, and the target document the `UIEvent` is related to.

In order to fully support this module, an implementation must also support the *Core* feature defined in the Document Object Model Level 2 Core specification.

DOM Level 2 Events

This specification defines the Document Object Model Level 2 Events, a platform- and language-neutral interface that gives to programs and scripts a generic event system. The Document Object Model Level 2 Events extends the Document Object Model Level 2 Core and on Document Object Model Level 2 Views.

The DOM Level 2 Event Model is designed with two main goals. The first goal is the design of a generic event system, which allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event. Additionally, the specification will provide standard modules of events for user interface control and document mutation notifications, including defined contextual information for each of these event modules.

The second goal of the event model is to provide a common subset of the current event systems used in DOM Level 0 browsers. This is intended to foster interoperability of existing scripts and content. It is not expected that this goal will be met with full backwards compatibility. However, the specification attempts to achieve this when possible.

DOM Level 2 Style

This specification defines the Document Object Model Level 2 Style Sheets and Cascading Style Sheets (CSS), a platform and language neutral interface that allows programs and scripts to dynamically access and update the content of style sheet documents. The Document Object Model Level 2 Style extends the Document Object Model Level 2 Core and on the Document Object Model Level 2 Views. The DOM Level 2 Style Sheet interfaces are base interfaces used to represent any type of style sheet. The expectation is that DOM modules that represent a specific style sheet language may contain interfaces that derive from these interfaces.

The DOM Level 2 Cascading Style Sheets (CSS) interfaces are designed with the goal of exposing CSS constructs to object model consumers. Cascading Style Sheets is a declarative syntax for defining presentation rules, properties and ancillary constructs used to format and render Web documents. This document specifies a mechanism to programmatically access and modifies the rich style and presentation control provided by CSS (specifically CSS level 2 [CSS2]). This augments CSS by providing a mechanism to dynamically control the inclusion and exclusion of individual style sheets, as well as manipulate CSS rules and properties.

The CSS interfaces are organized in a logical, rather than physical structure. A collection of all style sheets referenced by or embedded in the document is accessible on the document interface.

DOM Level 2 Traversal and Range

This specification defines the Document Object Model Level 2 Traversal and Range, platform- and language-neutral interfaces that allow programs and scripts to dynamically traverse and identify a range of content in a document. The Document Object Model Level 2 Traversal and Range specification extends the Document Object Model Level 2 Core specification.

The DOM Level 2 Traversal and Range specification is composed of two modules. The two modules contain specialized interfaces dedicated to traversing the document structure and identifying and manipulating a range in a document.

DOM Level 2 HTML

This specification defines the Document Object Model Level 2 HTML, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content and structure of [HTML 4.01] and [XHTML 1.0] documents. The Document Object Model Level 2 HTML extends the Document Object Model Level 2 Core and is not backward compatible with DOM Level 1 HTML.

This section extends the DOM Level 2 Core API [DOM Level 2 Core] to describe the objects and the methods specific to HTML documents [HTML 4.01], and XHTML documents [XHTML 1.0]. In general, the functionality needed to manipulate hierarchical document structures, elements, and attributes will be found in the core section; functionality that depends on the specific elements defined in HTML will be found in this section.

The goals of the HTML-specific DOM API are as follows:

- To specialize and add functionality that relates specifically to HTML documents and elements.
- To address issues of backwards compatibility with the DOM Level 0.
- To provide convenience mechanisms, wherever appropriate, for common and frequent operations on HTML documents.

The key differences between the core DOM and the HTML application of DOM is that the HTML Document Object Model exposes a number of convenience methods and properties that are consistent with the existing models and are more appropriate to script writers. In many cases, these enhancements are not applicable to general DOM because they rely on the presence of a predefined DTD. The transitional or frameset DTD for HTML 4.01, or the XHTML 1.0 DTDs are assumed. Interoperability between implementations is only guaranteed for elements and attributes that are specified in the HTML 4.01 and XHTML 1.0 DTDs.

More specifically, this document includes the following specializations for HTML:

- An `HTMLDocument` interface derived from the core Document interface, `HTMLDocument` specifies the operations and queries that can be made on a HTML document.
- An `HTMLElement` interface derived from the core Element interface, `HTMLElement` specifies the operations and queries that can be made on any HTML element. Methods on `HTMLElement` include those that allow for the retrieval and modification of attributes that apply to all HTML elements.
- Specializations for all HTML elements, which have attributes that extend beyond those specified in the `HTMLElement` interface. For all such attributes, the derived interface for the element contains explicit methods for setting and getting the values.

The DOM Level 2 includes mechanisms to access and modify style specified through CSS and define an event model that can be used with HTML documents.

DOM Level 3

The DOM Level 3 will extend Level 2 by finishing support for XML 1.0 with namespaces (alignment with the XML Infoset and support for XML Base) and will extend the user interface events (keyboard). It will also add abstract schemas support (for DTDs, XML Schema), the ability to load and save a document or an abstract schema, explore further mixed markup vocabularies and the implications on the DOM API ("Embedded DOM"), and support XPath.

As mentioned earlier that the level consists of six specifications, but here we'll discuss only those specification which are recommended, i.e. they are in final stage. These specifications are discussed here.

DOM Level 3 Core

This specification defines the Document Object Model Core Level 3, a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The Document Object Model Core Level 3 extends the Document Object Model Core Level 2 [DOM Level 2 Core].

This version enhances DOM Level 2 Core by completing the mapping between DOM and the XML Information Set [XML Information Set], including the support for XML Base [XML Base], adding the ability to attach user information to DOM nodes or to bootstrap a DOM implementation, providing mechanisms to resolve namespace prefixes or to manipulate ID attributes, giving to type information, etc.

This specification defines a set of objects and interfaces for accessing and manipulating document objects. The functionality specified (the Core functionality) is sufficient to allow software developers and Web script authors to access and manipulate parsed HTML and XML content inside conforming products. The DOM Core API also allows creation and population of a Document object using only DOM API calls. A solution for loading a Document and saving it persistently is proposed in [DOM Level 3 Load and Save].

DOM Level 3 Load and Save

This specification defines the Document Object Model Load and Save Level 3, a platform- and language-neutral interface that allows programs and scripts to dynamically load the content of an XML document into a DOM document and serialize a DOM document into an XML document; DOM documents being defined in [DOM Level 2 Core] or newer, and XML documents being defined in [XML 1.0] or newer. It also allows filtering of content at load time and serialization time.

DOM Level 3 Validation

This specification defines the Document Object Model Validation Level 3, a platform- and language-neutral interface. This module provides guidance to programs and scripts to dynamically update the content and the structure of documents while ensuring that the document remains valid, or to ensure that the document becomes valid.

This module provides Application Programming Interfaces (APIs) to guide construction and editing of XML documents. Examples of such guided editing are queries that combine questions like "what does the schema allow me to insert/delete here?" and "if I insert/delete here, will the document still be valid?"

To aid users in the editing and creation of XML documents, other queries may expose different levels of details, e.g. all the possible children, lists of defined symbols of a given kind. Some of these queries would prompt check and warn users if they are about to conflict with or overwrite such data.

Finally, users would like to validate an edited or newly constructed document before serializing it or passing it to other users. They may edit, come up with an invalid document, and then edit again to result in a valid document. During this process, these APIs can allow the user to check the validity of the document or sub tree on demand. If necessary, these APIs can also require that the document or sub tree remain valid during this editing process via the `DocumentEditVal.continuousValidityChecking` flag.

A `DOM` application can use the `hasFeature(feature, version)` method of the `DOMImplementation` interface to determine with parameter values `Validation` and `3.0`, respectively, whether or not these interfaces are supported by the implementation. This implementation is dependent on [`DOM` Level 2 Core] and the [`DOM` Level 3 Core] `DOMConfiguration` interface.

After discussing about JavaScript and DOM, let's create a Web application using JavaScript. The application created in the following section, does not use AJAX features.

Creating a JavaScript Application without AJAX

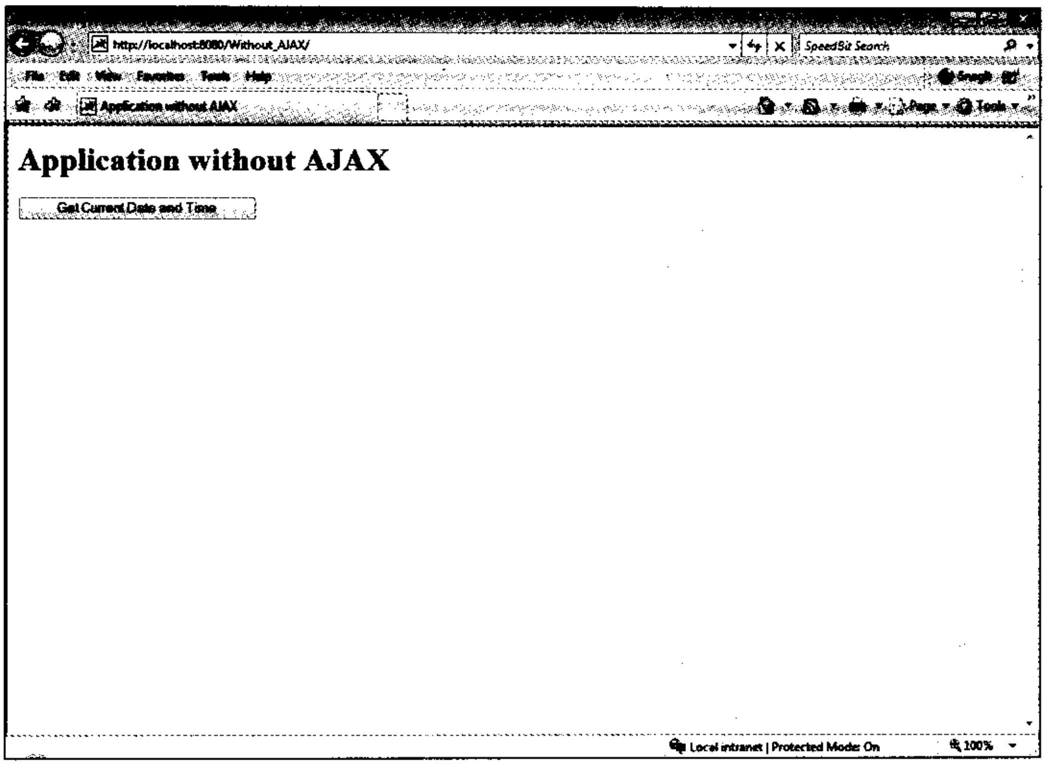
In this application, we will send a POST request to the server without using AJAX technique. The application demonstrates that from sending the request to the server, the server performing the request processing and returning the HTML page to the client, the client has to keep waiting. The application sends a request for the system's current date and time to the server. The application starts with a HTML page, index.html. The HTML page displays a button labeled "Get Current Date and Time" and when the client presses this button it sends a POST request to the server for accessing the current date and time from the file date.jsp.

Listing 29.1 shows the source code for the index.html page (you can find this file in the Code/AJAX/Chapter 29/Without_AJAX folder on the CD):

Listing 29.1: index.html

```
<html>

<head>
  <title>Application without AJAX</title>
</head>

<body>
  <H1>Application without AJAX</H1>
  <form action = date.jsp method=POST>
  <input type = "submit" value = "Get Current Date and Time">
  </form>
</body>

</html>
```

When you open the HTML file, index.html, it is displayed like the one shown in Figure 29.3:


http://localhost:8080/Without_AJAX/
SpeedBit Search
File Edit View Favorites Tools Help
Application without AJAX
Page Tools
Local intranet | Protected Mode: On
100%

Application without AJAX

Get Current Date and Time

Figure 29.3: Accessing Current Date and Time without using AJAX

When you click on the button "Get Current Date and Time", the HTML form sends a POST request to the server for accessing current date and time from the file date.jsp.

Listing 29.2 shows the source code for date.jsp file (you can find this file in the Code/AJAX/Chapter 29/Without_AJAX folder on the CD):

Listing 29.2: date.jsp

```
<%@page contentType="text/html" import="java.util.*" %>
<html>
<body>
<p>&nbsp;</p>
<div align="left">
<table border="0" cellpadding="0" cellspacing="0" width="460" bgcolor="[illegible]">
<tr>
<td width="100%"><font color="#000000">Without AJAX</font></td>
</tr>
<tr>
<td width="100%"><b>&nbsp;Current Date and time is:&nbsp;<font color="#FF0000">
<%= new java.util.Date()%>
</font></b></td>
</tr>
</table>
</div>
</body>
</html>
```

The JSP page displays the current date and time when you click the Get Current Date and Time button of the index.html page, as shown in Figure 29.4:

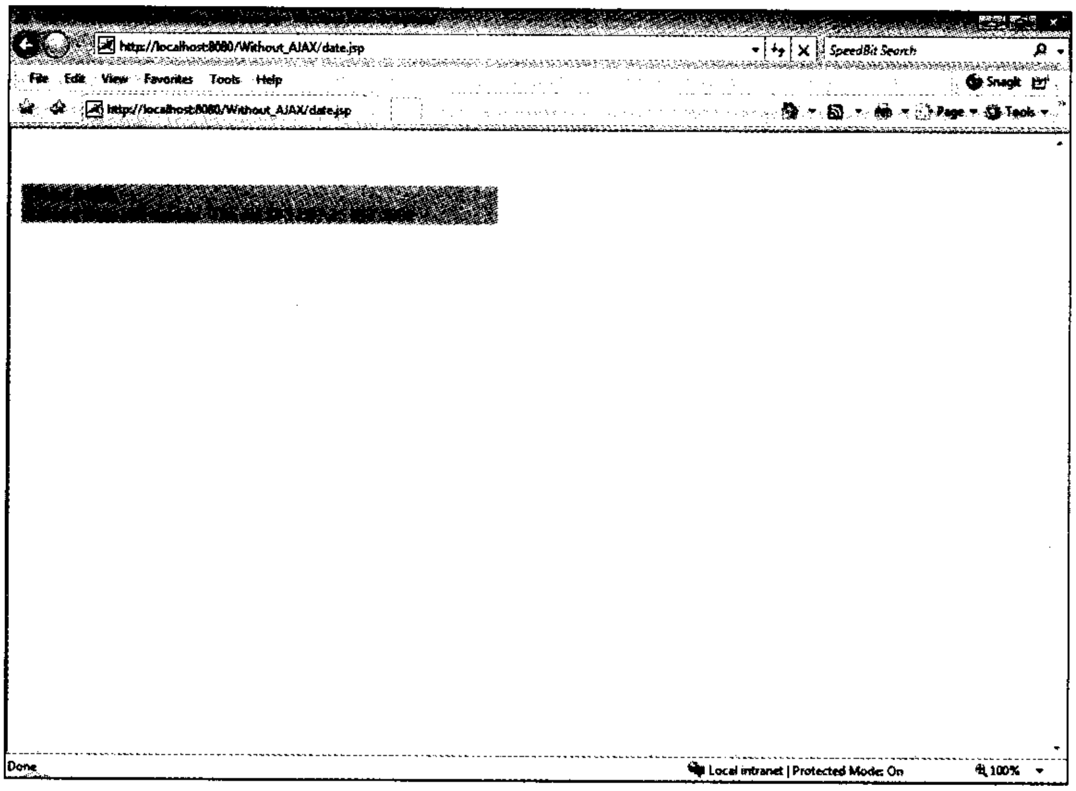

http://localhost:8080/Without_AJAX/date.jsp
SpeedBit Search
File Edit View Favorites Tools Help
Snagit
Page
Tools
Done
Local intranet | Protected Mode: On
100%

Figure 29.4: Showing Current Date and Time without using AJAX

The date.jsp page displays the current date and time without using any AJAX feature, as shown in Figure 29.4.

You have learned creating Web application using JavaScript, in this section. Let's now discuss creating Web applications by using JavaScript and AJAX, in the following section.

JavaScript and AJAX

With AJAX, your `JavaScript` can communicate directly with the server, using the `JavaScript` object. With this object, your `JavaScript` can trade data with a Web server, without reloading the page. AJAX uses asynchronous data transfer (HTTP requests) between the browser and the Web server, allowing web pages to request small bits of information from the server, instead of whole pages. The AJAX technique makes Internet applications smaller, faster, and more user-friendly. AJAX is a browser technology independent of Web server software.

`JavaScript` plays an important role in the AJAX technique. Using `JavaScript` technology, an HTML page can asynchronously make calls to the server from which it loads and fetches contents that may be formatted as XML documents, HTML content, plain text, or `JavaScript` Object Notation (JSON). The `JavaScript` technology may then use the content to update or modify the Document Object Model (DOM) of the HTML page. The term Asynchronous `JavaScript` Technology and XML (AJAX) has emerged recently to describe this interaction model.

AJAX is not a new technique as `JavaScript` and XML are being already in use for a long time. What has changed recently is the inclusion of support for the XMLHttpRequest object in the JavaScript runtimes of the mainstream browsers. Although this object is not specified in the formal JavaScript technology specification, all of today's mainstream browsers support it.

What makes AJAX-based clients unique is that the client contains page-specific control logic embedded as JavaScript technology. The page interacts with the JavaScript technology based on events, such as the loading of a document, a mouse click, focus changes, or even a timer. AJAX interactions allow for a clear separation of presentation logic from the data. An HTML page can pull in bite-size pieces to be displayed.

With an HTTP request, a web page can make a request to, and get a response from a Web server - without reloading the page. The user will stay on the same page, and he or she will not notice that scripts might request pages, or send data to a server in the background. This is the most important functionality that JavaScript provides to AJAX. Let's create a Web application using JavaScript and AJAX in the following section.

Creating a JavaScript Application with AJAX

So far, we've covered the basics of JavaScript and discussed how to get a web page to call JavaScript functions in response to user events. This covers a third of what you need to know to create an AJAX application. In this application we will implement how JavaScript uses the XMLHttpRequest object for sending the request to the server and receiving response from the server. The application starts with a HTML page, datetime.html, that displays a button labeled "Get Current Date and Time".

Listing 29.3 shows the source code for thedatetime.html page (you can find this file in the Code/AJAX/Chapter 29/Simple_AJAX folder on the CD):

Listing 29.3: datetime.html

```
<html>
<head>
   <title>JavaScript-Ajax Application</title>

   <script language = "javascript">
   var XMLHttpRequestObject = false;

   if (window.XMLHttpRequest) {
         XMLHttpRequestObject = new XMLHttpRequest();
   } else if (window.ActiveXObject) {
         XMLHttpRequestObject = new
         ActiveXObject("Microsoft.XMLHTTP");
   }

   function getDateTime(datetimefile, divID)
   {
         if(XMLHttpRequestObject) {
                  var obj = document.getElementById(divID);
                  XMLHttpRequestObject.open("POST", datetimefile);

                  XMLHttpRequestObject.onreadystatechange = function()
         {
         if (XMLHttpRequestObject.readyState == 4 &&
         XMLHttpRequestObject.status == 200) {
                  obj.innerHTML =
                  XMLHttpRequestObject.responseText;
                  }
         }

         XMLHttpRequestObject.send(null);
   }

   }
   </script>
```

```
</head>
<body>
   <H1>JavaScript-Ajax Application</H1>
   <form>
   <input type = "button" value = "Get Current Date and Time"
   onclick = "getDateTime('date.jsp', 'targetDiv')">
   </form>
   <div id="targetDiv">
   </div>
</body>
</html>
```

When you open the datetime.html page, it is displayed, as shown in Figure 29.5:

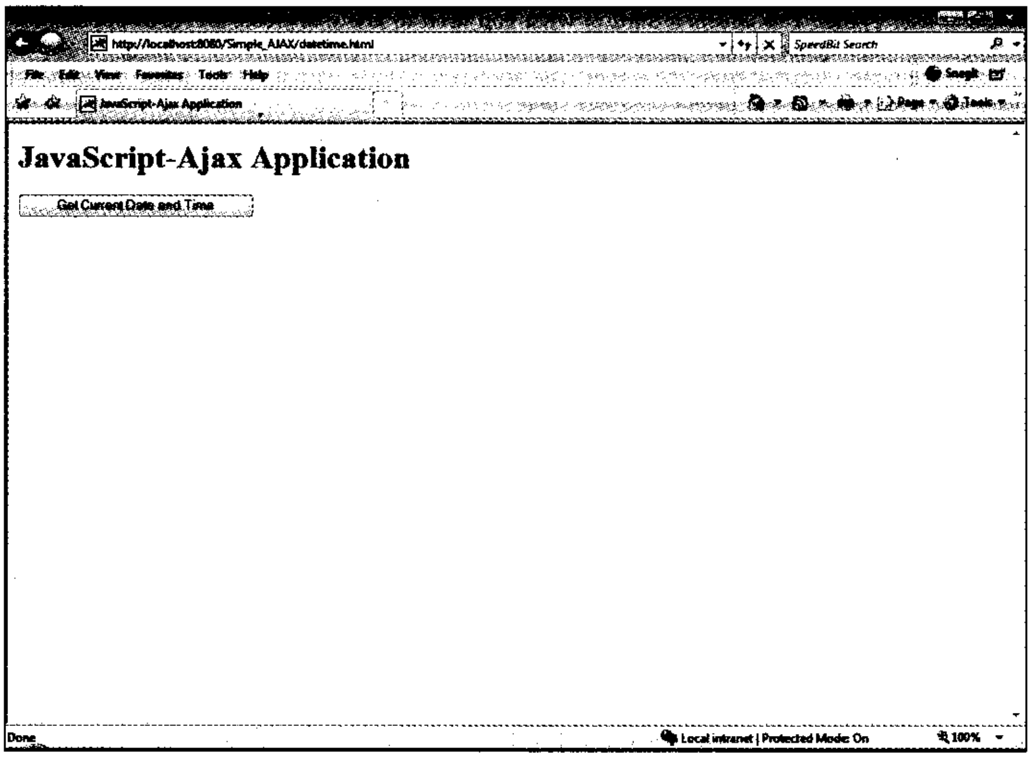

http://localhost:8080/Simple_AJAX/datetime.html
SpeedBit Search
File Edit View Favorites Tools Help
JavaScript-Ajax Application
JavaScript-Ajax Application
Get Current Date and Time
Done
Local intranet | Protected Mode: On
100%

Figure 29.5: JavaScript-AJAX Application

When you click on the Get Current Date and Time button , the XMLHttpRequest sends a POST request to the server for accessing current date and time from the file date.jsp.

Listing 29.4 shows the source code of the date.jsp file (you can find this file in the Code/AJAX/Chapter 29/Simple_AJAX folder on the CD):

Listing 29.4: date.jsp

```
<%@page contentType="text/html" import="java.util.*" %>
<html>
<body>
<p>&nbsp;</p>
<div align="left">
<table border="0" cellpadding="0" cellspacing="0" width="460" bgcolor="#ADADDE">
```

```
<tr>
<td width="100%"><font size="6" color="#000000">AJAX Date</font></td>
</tr>
<tr>

<td width="100%"><b>&nbsp;Current Date and time is:&nbsp; <font color="#FF0000">
<%= new java.util.Date()%>
</font></b></td>
</tr>
</table>

</div>

</body>

</html>
```

After clicking the Get Current Date and Time button on the datetime.html page, the HTML page displays the current date and time, as shown in Figure 29.6:

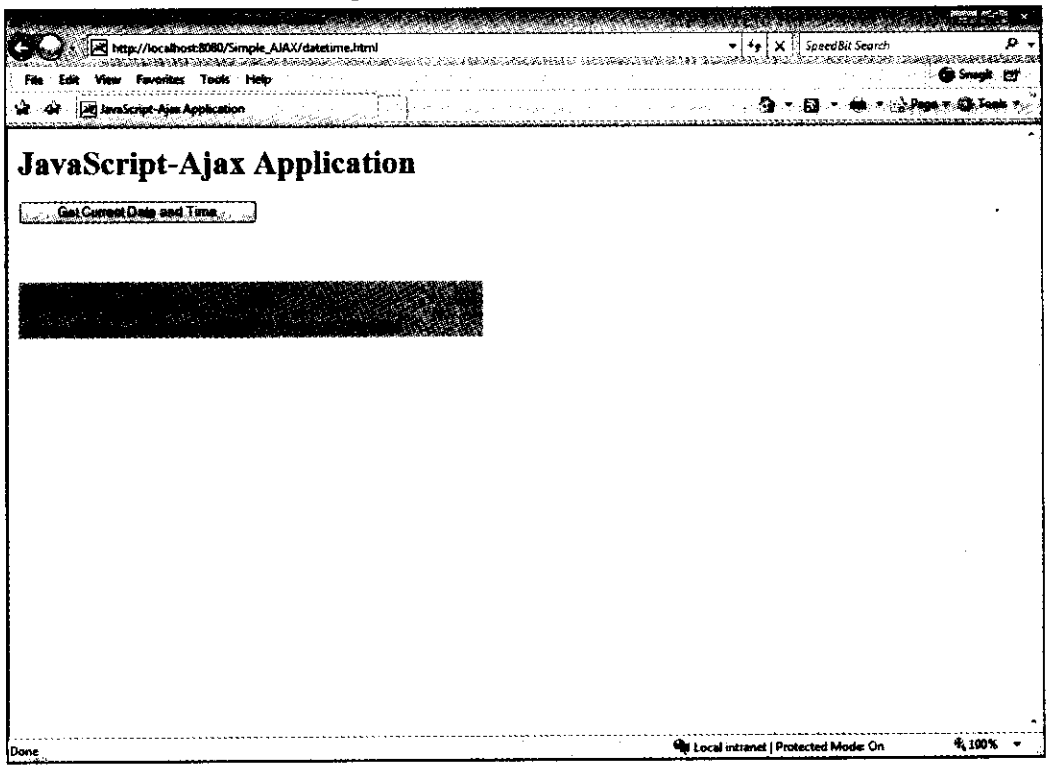

http://localhost:8080/Simple_AJAX/datetime.html
SpeedBit Search
File Edit View Favorites Tools Help
JavaScript-Ajax Application
JavaScript-Ajax Application
Get Current Date and Time
Done
Local intranet | Protected Mode: On
100%

Figure 29.6: AJAX Application Showing Current Date and Time

In this application we first create an XMLHttpRequest object according to the browser is used. Next we define a JavaScript function getDateTime () which uses the XMLHttpRequest for sending asynchronous request to the server. In this function, we call the XMLHttpRequest object's open () function and passed the date.jsp page as a parameter to this function. The XMLHttpRequest object sends a asynchronous POST request for date.jsp file to the server. After opening the XMLHttpRequest object, the XMLHttpRequest object has the property named onreadystatechange, which allows handling the asynchronous loading operations. If this property is assigned to the name of the JavaScript function then, this function will be called each time the XMLHttpRequest object's state changes.

Now finally when the XMLHttpRequest object is in its ready state and the status is equal to 200, then the data is fetched. The status 200 refers to the 'Ok' state of the XMLHttpRequest object. So, to make sure that the data is completely downloaded, we check the value of the status property with 200. Finally when the data is downloaded, the data is retrieved in either the standard HTML or the XML format. If responseText property is used then the data is retrieved in the standard HTML format. But if your data is formatted as XML, then responseXML property is used. We are using responseText since our data is in standard HTML format that is returned by the date.jsp page.

Now after retrieving the data, in order to display the data on the Web page, you can assign that text to the <div> element, whose ID is targetDiv in the web page and whose name was passed to the getDateTime() function.

Summary

In this chapter, you have learned about `JavaScript` in detail. You learned about the reasons behind the evolution of the `JavaScript`, its advantages,, the syntax of using `JavaScript`, the data types, variables, operators, functions, loops, and objects present in `JavaScript` along with the Document Object Model (`DOM`) and its various levels. The chapter also discussed creating a Web application using JavaScript. Further the chapter discussed about use of JavaScript with AJAX. Towards the end of the chapter, you learned creating Web applications using JavaScript and AJAX.

The next chapter will focus on XMLHttpRequest objects, which is another essential part of `AJAX`.

Quick Revise

Q1. Explain the relationship of JavaScript and AJAX.

Ans: `JavaScript` is an essential part of AJAX. The `JavaScript` code in an AJAX application sends the requests of the client to be processed by the server, but doesn't wait for an answer. Even better, `JavaScript` can also work with the server's response that comes in the form of XML, instead of reloading the entire page when the server is finished with your request. In this case, the `JavaScript` processes the XML document according to `DOM` model.

Q2. List the conditional statements used in JavaScript.

Ans: Following are the conditional statements used in JavaScript:

- if statement
- if-else statement
- switch statement
- while loop
- do-while loop
- for loop

Q3. List the JavaScript object types.

Ans: The objects available in the `JavaScript` can be divided into following types:

- Built-in objects
- Browser objects
- User-defined objects

Q4. List the properties of Windows object of JavaScript.

Ans: Some of the properties available with the window object are as follows:

- `closed`
- `defaultStatus`
- `document`
- `history`
- `location`

- `name`
- `opener`
- `outerheight`
- `outerwidth`
- `self`
- `status`

Q5. List the levels of DOM.

Ans: The following are the levels of DOM:

- **Level 0**—The application supports an intermediate DOM, which existed before the creation of DOM Level 1. Examples include the DHTML Object Model or the Netscape intermediate DOM. Level 0 is not a formal specification published by the W3C, but rather a shorthand that refers to what existed before the standardization process.
- **Level 1**—It includes the Navigation of DOM (HTML and XML) document (tree structure) and content manipulation (includes adding elements). HTML-specific elements are included as well.
- **Level 2**—XML namespace support, filtered views and events.
- **Level 3** - This level consists of 6 different specifications:
  - DOM Level 3 Core
  - DOM Level 3 Load and Save
  - DOM Level 3 XPath
  - DOM Level 3 Views and Formatting
  - DOM Level 3 Requirements
  - DOM Level 3 Validation, which further enhances the DOM

Q6. Define DOM Level 2 Core.

Ans: The Document Object Model Level 2 Core extends the Document Object Model Level 1 Core. The DOM Level 2 Core is made of a set of core interfaces to create and manipulate the structure and contents of a document. The Core also contains specialized interfaces dedicated to XML.

Q7. What are the goals of DOM Level 2 Events?

Ans: The DOM Level 2 Event Model is designed with two main goals. The first goal is the design of a generic event system, which allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event. Additionally, the specification will provide standard modules of events for user interface control and document mutation notifications, including defined contextual information for each of these event modules.

The second goal of the event model is to provide a common subset of the current event systems used in DOM Level 0 browsers. This is intended to foster interoperability of existing scripts and content. It is not expected that this goal will be met with full backwards compatibility.

Q8. What is the purpose of CSS?

Ans: Cascading Style Sheets is a declarative syntax for defining presentation rules, properties and ancillary constructs used to format and render Web documents.

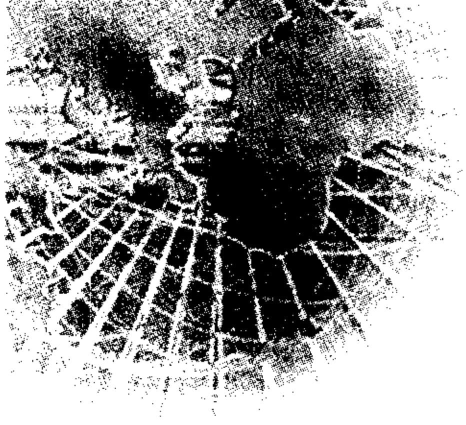
Q9. What are the goals of HTML specific DOM API?

Ans: The goals of the HTML-specific DOM API are as follows:

- To specialize and add functionality that relates specifically to HTML documents and elements.
- To address issues of backwards compatibility with the DOM Level 0.
- To provide convenience mechanisms, wherever appropriate, for common and frequent operations on HTML documents.

Q10. Define DOM Level 3 Load and Save.

Ans: DOM Level 3 Load and Save is a platform- and language-neutral interface that allows programs and scripts to dynamically load the content of an XML document into a DOM document and serialize a DOM document into an XML document; DOM documents being defined in [DOM Level 2 Core] or newer, and XML documents being defined in [XML 1.0] or newer. It also allows filtering of content at load time and serialization time.



30

Asynchronous data transfer with XMLHttpRequest

The key to AJAX technique is an object called `XMLHttpRequest`, which is used by JavaScript. The XMLHttpRequest object can send HTTP requests, receive responses, and parse them as XML. Since Chapter 28, we have been using `XMLHttpRequest` object. Now this chapter discusses the history and implementation of `XMLHttpRequest` object in detail. Though in the first chapter a brief introduction of `XMLHttpRequest` is provided, but this chapter provides the depth study of it.

The chapter first discusses the history of `XMLHttpRequest` object. `XMLHttpRequest` is the API which can be used by JavaScript, VBScript and other Web browser scripting languages, to transfer the XML and other text data to and from the Web server by using HTTP. Apart from the data in XML format, the `XMLHttpRequest` object can be used to fetch the data in HTML, JSON, or plain text format. `XMLHttpRequest` is an important part of the Ajax web development technique, and it is used by many websites to implement interactive, responsive and dynamic websites or Web applications.

XMLHttpRequest Object

The concept of `XMLHttpRequest` object was developed by Microsoft as a part of Outlook Web Access 2000 (as the server-side API call). The Outlook Web Access 2000 was the Outlook Web-mail service which allows users to access the e-mail functionality. This was done by allowing the application to issue its own client-side HTTP requests. Then Microsoft quickly drafted it into IE5 and this is accessible through Jscript, VBScript and the other scripting languages supported by IE browsers. Therefore, the life of `XMLHttpRequest` object started as an ActiveXControl in Internet Explorer 5.

The ActiveXControl could run only on the Internet Explorer (IE). The `XMLHttpRequest` object also became popular in the other Web browsers, like Mozilla 1.0, Netscape 7, Safari 1.2, and Opera 7.60. The XMLHttpRequest object varies from browser to browser. The different version of IE uses different XMLHttpRequest object and the XMLHttpRequest object used in the other browsers, which run as a part of the window object, is also different from IE. The browser-detecting script should be used, and then the instance of the `XMLHttpRequest` should be made. The W3C standards organization took over the implementation of the `XMLHttpRequest` object with the intent of creating the common set of properties and methods which can be used all over the browsers.

As a result, the `XMLHttpRequest` is now included in IE 7 as a native object (which means it works in the same way as it does on Firefox), and the `XMLHTTP` object is still present as an ActiveXControl. With IE 6 being an integral part of the Windows XP operating system, you are likely to have to make arrangements for it in your applications for a long time yet.

The major purpose of `XMLHttpRequest` object is to be able to use HTML and script to connect directly with the data layer, which is stored on the server. The advantage of `XMLHttpRequest` object is that there is no need to send the page and refresh it because the changes to the data are immediately reflected in the web page displayed by the Web browser. With the `XMLHttpRequest` object, Microsoft Internet Explorer clients or the users of other Web browsers can retrieve and submit XML data directly to a Web server without reloading the page. To convert XML data into HTML content, use the client-side XML DOM or Extensible Stylesheet Language Transformations (XSLT) to compose HTML elements for presentation.

As discussed in Chapter 28, the code for creating an `XMLHttpRequest` object in both Firefox and IE is given here. Here's the code for creating an `XMLHttpRequest` for IE7, Firefox, Safari and Opera:

```
var xmlRequest= new XMLHttpRequest();
```

But, if the Web browsers are IE5 or IE6, then the following code can be used to create an object of `XMLHttpRequest`:

```
var xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
```

With the help of the preceding code, the `XMLHttpRequest` object can be created depending upon the browser used, whether it is Firefox or Internet Explorer 6. This shows that depending upon the browser, the `XMLHttpRequest` object is created. Here's the complete code for detecting the browser type and then creating an `XMLHttpRequest` object:

```
function getReq()
{
    if (window.XMLHttpRequest)
    {
          xmlRequest = new XMLHttpRequest();
    }
    else if (window.ActiveXObject)
    {
          try
          {
                xmlRequest = new ActiveXObject("Msxml2.XMLHTTP");
          }
          catch(e)
          {
                try {
                      xmlRequest = new ActiveXObject("Microsoft.XMLHTTP");
                }
                catch(e) {
                      xmlRequest = false;
                }
          }
    }
}
```

In the preceding code snippet, the XMLHttpRequest object is xmlRequest. Depending upon the browser you are using, XMLHttpRequest object will be created.

Creating an object on the web page will not produce anything visible. The XMLHttpRequest object can be used in either of the following two ways:

- Synchronously
- Asynchronously

Synchronous and Asynchronous Pages

After making the instance of the XMLHttpRequest object, depending upon the browser in use, it is important to decide whether the loading of the page or the data will be synchronous or asynchronous. The XMLHttpRequest object can be used in calling the web page in either synchronous or asynchronous mode. If the XMLHttpRequest object is used synchronously, then after sending the request to the server, the user has to wait till the response is received from the server. Therefore, in the synchronous request, the XMLHttpRequest object works in the following pattern:

1. Create the object
2. Create a request
3. Send the request
4. Hold the processing until you get the response

Here's the code for getting the file in synchronous mode:

```
var xmlRequest = new XMLHttpRequest();
xmlRequest.open('GET', 'address.xml', false);
xmlRequest.send(null);
var xmlRequest = xmlRequest.responseXML;
```

In order to get the file in a synchronous way, according to the steps given earlier, an object is created through a new operator of JavaScript. The first step in the preceding code helps in creating the instance of XMLHttpRequest object by using the JavaScript new operator. Then the open() method is invoked by using the request method, GET, a destination URL, and a Boolean value 'false', indicating that the request is synchronous. Thirdly the send method is invoked. Finally, the responseXML, an XML document, is assigned to a variable. Moreover, instead of responseXML, responseText can also be used.

There are only a few differences between asynchronous and synchronous request. In asynchronous request, the following is the pattern:

1. Create the object
2. Set the `readystatechange` event to trigger a specific function
3. Check the `readyState` property, to see if the data is ready. If it is not, then check it again after an interval.
4. If the state becomes ready, then follow the next step
5. Open Request
6. Send Request
7. Continue the processing. Interruption is done only when the response is received

Now let's quickly analyze how synchronous and asynchronous requests vary from each other. One of the differences is to set the Boolean property in the open method as true. True means that the script continues to run after the `send()` method, without waiting for a response from the server. On the other hand, false means that the script waits for a response before continuing script processing. The true value set for async parameter, indicates that the request is asynchronous. Moreover in asynchronous, instead of waiting for the response from the send method, the processing continues. The following code shows how the request is made asynchronously:

```
var xmlRequest = new XMLHttpRequest();
xmlRequest.onreadystatechange = asyncHandler;
xmlRequest.open('GET', 'address.xml', true);
xmlRequest.send(null);
function asyncHandler()
{
  if(xmlRequest.readyState == 4)
  var objXML = xmlRequest.responseXML;
}
```

The entire processing of the `readystatechange` event is performed behind the scenes and it enables us to use `XMLHttpRequest` object asynchronously. The major thing to note here is that the `XMLHttpRequest` object with the help of the `readyState` property, informs when the loading of data is going to be finished, as its properties and methods cannot be used unless the loading completes.

XMLHttpRequest Properties

The `XMLHttpRequest` object has six properties and six methods to represent the request of XML, plain text or JOSN formatted data, through HTTP. The following are the properties of `XMLHttpRequest` object:

- onreadystatechange Property
- readyState Property
- responseText Property
- responseXML Property
- status Property
- statusText Property

Now let's explore more about these properties.

onreadystatechange Property

This is an event handler for an event which triggers at every state change. This property sets the method to be called on every state change. This is usually the event handler for the asynchronous callback. This property sets or retrieves the event handler for the asynchronous requests made.

readyState Property

This property defines the current state of the `XMLHttpRequest`. It defines at what point the `XMLHttpRequest` object can send or receive the data. The following are the possible values indicating the various states of the `XMLHttpRequest`:

- **0= Uninitialized**–This is the stage when the object has to be created but has not been initialized. In other words, the open method has to be called.

- **1= Loading/Open**—This is the stage when the object has been created and initialized, but the send method has not been called.
- **2= Loaded/Sent**—This is the stage when the `send()` method is called, but the send method is waiting for the return of the status code and the headers.
- **3= Interactive/Receiving**—This is the stage in which some data has been received, but not all. The properties of the object will not be used to view the partial results because status and response headers are not fully available.
- **4= Complete/Loaded**—This is the final stage where the whole data is received and the complete data is available. This is the `readyState` property code which needs to be checked.

In the asynchronous callback handler, the state is checked to see whether the `readyState` is complete or equal to 4. Once the data has been loaded completely, the other properties and methods can be used to get back the data from the response to the request made.

responseText Property

This property of `XMLHttpRequest` object returns the response in the form of a string. When the `readyState` value is 0, 1 or 2, then the `responseText` contains an empty string. But when the `readyState` value changes to 3 (Receiving), the `responseText` contains the incomplete response received by the client. Finally, when the `readyState` changes to 4 (Loaded), the `responseText` contains the complete response received by the client.

responseXML Property

This property of `XMLHttpRequest` object returns the response as XML. This property returns an XML document object, which can be examined and parsed by using W3C DOM node tree methods and properties. The `responseXML` property represents the XML response When the complete HTTP response has been received (when the `readyState` is 4), then the Content-Type header specifies the MIME (media) type as text/xml, application/xml, or ends in +xml. If the Content-Type header does not contain one of these media types, then the `responseXML` contains null value. Moreover, the `responseXML` value is also null when the `readyState` value is not equal to 4.

The `responseXML` property value is an object of type Document interface, and represents the parsed document. If the document cannot be parsed, then the `responseXML` value is null.

status Property

This property represents the HTTP status code and its datatype is of short type. The status attribute is available only when the readyState value is either 3 or 4. If the status value is accessed when the readyState value is less than 3 then, in that case, an exception arises. If the status property has the value 200, it indicates the successful operation. Table 30.1 lists a complete summary of the HTTP Status code:

Table 30.1: Http Status Code

HTTP Status code	Http Status text
100	Continue
101	Switching Protocols
200	OK
201	Created
202	Accepted
203	Non-Authoritative Information
204	No Content
206	Partial Content
300	Multiple Choice

Table 30.1: Http Status Code

HTTP Status code	Http Status text
301	Move Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
306	Unused
307	Temporary Redirect
400	Bad Request
402	Payment Required
403	Forbidden
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Request Entity Too Large
414	Request URI Too Long
415	Unsupported Media Type
416	Request Range Not Satisfiable
417	Expectation Failed
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported

The status property returns the HTTP status code of the request.

statusText Property

The `statusText` attribute represents the HTTP status code text and similar to status property, this property also provides the `statusText` when the `readyState` value is either 3 or 4.

Using XMLHttpRequest Properties

The following code snippet provides the implementation of the previously discussed `XMLHttpRequest` object's properties:

```
xmlRequest.open("GET", "address.xml",true);
xmlRequest.onreadystatechange = function()
{
  if (xmlRequest.readyState == 4)
  {
       if (xmlRequest.status == 200)
       {
              var emp = request.responseText;
              alert(emp);
       }
  }
}
```

In the preceding code snippet, the `xmlRequest`, `XMLHttpRequest` object created before in the chapter, uses its properties. Firstly the request is made for `address.xml` file. The first property is `onreadystatechange`, and whenever the state of `xmlRequest` changes the function defined in JavaScript is called. This function, using `readyState` property of `xmlRequest`, checks whether the value of `readyState` is 4. If the value of `readyState` is 4, then the status of `xmlRequest` is checked. And if the value of the status is 200, then by using the `responseText` property, the entire data in the `address.xml` file is retrieved and displayed as a text string. If, instead of `responseText` property, the `responseXML` is used, then the response received is in the form of XML document object.

The above implementation should have cleared the `XMLHttpRequest` object's properties.

XMLHttpRequest Methods

The `XMLHttpRequest` object also provides various methods to initiate and process HTTP requests. The following is the discussion for the `XMLHttpRequest` methods.

abort() Method

The `abort()` method is used to cancel the current `XMLHttpRequest` and reset the object to the uninitialized state.

open() Method

The `open()` method requires the method parameter to specify how the request should be sent. The following are the various parameters passed with the `open()` method:

- **DOMString method**—The various HTTP methods can be `GET`, `POST`, `PUT`, `DELETE` or `HEAD`. If the data is sent to the server then the `POST` method should be used. But, in order to retrieve the data from the server, the `GET` method is used.
- **DOMString uri**—The uri parameter specifies the server URI to which the `XMLHttpRequest` object sends the request. The uri resolves to an absolute URI using `window.document.baseURI` property. In other words, the absolute URIs can also be used that will be resolved in the same way as the browsers resolve the relative URIs.
- **Boolean async**—The async property specifies whether the request is made synchronously or asynchronously. If the request is asynchronous, then the default value of the async parameter is true, else if it is a synchronous request, then the value is false.
- **DOMString username**—The username and password of the servers, which require the authentication, are also passed in open() method. The username is an optional parameter.
- **DOMString password**—Similar to the username parameter, for server authentication, the password parameter can also be passed. It is also an optional parameter.

After calling the `open()` method, the `XMLHttpRequest` object sets its `readyState` value to 1 and resets the `responseText`, `responseXML`, status, and `statusText` properties to its initial values.

NOTE

The XMLHttpRequest object resets these values if the open() method is called and the readyState value is equal to 4.

send() Method

The `send()` method sends the HTTP request to the server and receives the response. After preparing the request by using the `open()` method, it can be send to the server using send() method. The request can be send to the server when the `readyState` value is 1, otherwise an exception is raised. The request is sent to the server by using the parameters provided in the `open()` method. The `send()` method returns immediately if the async parameter has the value set as true, allowing the other client script processing to be continued.

The `XMLHttpRequest` object sets the `readyState` value to 2 (Sent) after the `send()` method has been called. When the server responds, before receiving the message body, if any, the `XMLHttpRequest` object sets `readyState` to 3 (Receiving). When the request has completed loading, it sets `readyState` to 4 (Loaded). For a request of type `HEAD`, it sets the `readyState` value to 4, immediately after setting it to 3. The `send()` method takes an optional parameter that may contain data of varying types. Typically, you use this to send data to the server using the `POST` method. You can explicitly invoke the `send()` method with null, which is the same as invoking it with no argument. For most other data types, set the Content-Type header using the `setRequestHeader()` method (explained here) before invoking the `send()` method. If the data parameter in the `send(data)` method is of type `DOMString`, encode the data as UTF-8. If data is of type Document then serialize the data by using the encoding specified by `data.xmlEncoding`, if supported. If data.xmlEncoding is not supported then UTF-8 encoding is used.

setRequestHeader() method

This method adds the custom HTTP headers to the request made. It sets the header to the request by adding the name and value pair to the http header to be sent. This method takes two parameters—the label and value.

getResponseHeader() method

This method returns the value of the specified HTTP header. It has a single parameter that is the name of the header.

getAllResponseHeaders() Method

This method returns the complete set of http headers as the string. This method returns null, if the value of `readyState` is neither 3 nor 4.

Apart from supporting various methods, the `XMLHttpRequest` objects also support multiple brwosers. Let's discuss AJAX behavior for different browser, in the following section.

Browser Differences

The `XMLHttpRequest` object is widely used for sending AJAX request. The reason behind its popularity is that it is easy to use in a compatible way across multiple browsers. The two major browsers, Internet Explorer and FireFox, provide the same basic API. The other browsers, like Opera and Safari, also support the same basic API, but only in their more recent versions.

While writing the cross browser, the first problem to overcome is that `XMLHttpRequest` is an `ActiveXObject` in Internet Explorer, whereas it is a normal JavaScript object in other browsers, like Mozilla, Opera, etc. The solution to this problem is to detect the browser and accordingly create the `XMLHttpRequest` object. The following code snippet detects the browser in use and then creates the `XMLHttpRequest`, accordingly:

```
function getReq()
{
  if (window.XMLHttpRequest)
  {
        xmlhttp=new XMLHttpRequest();
  }
  else if (window.ActiveXObject)
  {
```

```
            try
            {
                    xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
            }
            catch(e)
            {
                    try
                    {
                            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
                    }
                    catch(e)
                    {
                            xmlhttp = false;
                    }
            }
    }
}
```

The overall pattern of the preceding code snippet is very simple and an instance of `XMLHttpRequest` is created, if the browser itself has `XMLHttpRequest` functionality. But if the browser has the `ActiveXControl`, then in the preceding code the `XMLHttpRequest` instance is made accordingly.

After discussing the concept of loading pages synchronously and asynchronously and properties and methods of the `XMLHttpRequest` object, let's create Web applications that load pages synchronously and asynchronously, in the following sections.

Reading a File Synchronously

We discussed earlier that if the async parameter of the `open ()` method of `XMLHttpRequest` is set false, and then the file is loaded synchronously. The following is the code in which the `helloAjax.xml` is loaded synchronously. By synchronous mode, we mean that the user will have to wait till the entire file is not loaded. In Listing 30.1 the `onreadystatechange` event does not fire since the file is read synchronously.

The code given in Listing 30.1 shows that the state of the `XMLHttpRequest` object is not alerted, but when the entire file is loaded, the data gets displayed (you can find hello.html file in the Code/AJAX/Chapter 30/Async folder on the CD):

Listing 30.1: The `hello.html File`

```
<html>

 <head>

  <title>Synchronously-Reading the File</title>

  <script type="text/javascript">
function getXHR()
{
  var RequestObject;
  if (window.XMLHttpRequest)
  {
  RequestObject = new XMLHttpRequest();
  } else if (window.ActiveXObject)
  {
  RequestObject = new ActiveXObject("Microsoft.XMLHTTP");
  }
  return RequestObject;
}
 function hello()
  {
   var requestObject = getXHR();
```

```
    requestObject.open("Get","helloAjax.xml",false);
    requestObject.send(null);
  requestObject.onreadystatechange=function()
  {
  alert(requestObject.readyState);
  }
    if(requestObject.status==200)
  {
  document.getElementById('message').innerHTML=  '<p>'  +  "The  ready  State  of  the
  XMLHttpRequest object is : " + requestObject.readyState + '</p><p>' ;
  var nodes=requestObject.responseXML.getElementsByTagName("name");
  for (i=0; i<nodes.length; i++)
  {
  document.getElementById('message').innerHTML += '<b>' + nodes[i].firstChild.nodeValue +
  '</b><br/>';
  }
    }
    else
  {
       alert ("Request failed");
    }
  }  </script>
</head>
<body>
  <h1>Synchronously-Reading the File</h1>
  <input type="button" value="Click Me" onclick="hello()"/>
  <p>
  <h3>CLICK the button above to synchronously read the file</h3></p>
   <DIV id="message"></DIV>
</body>
</html>
```

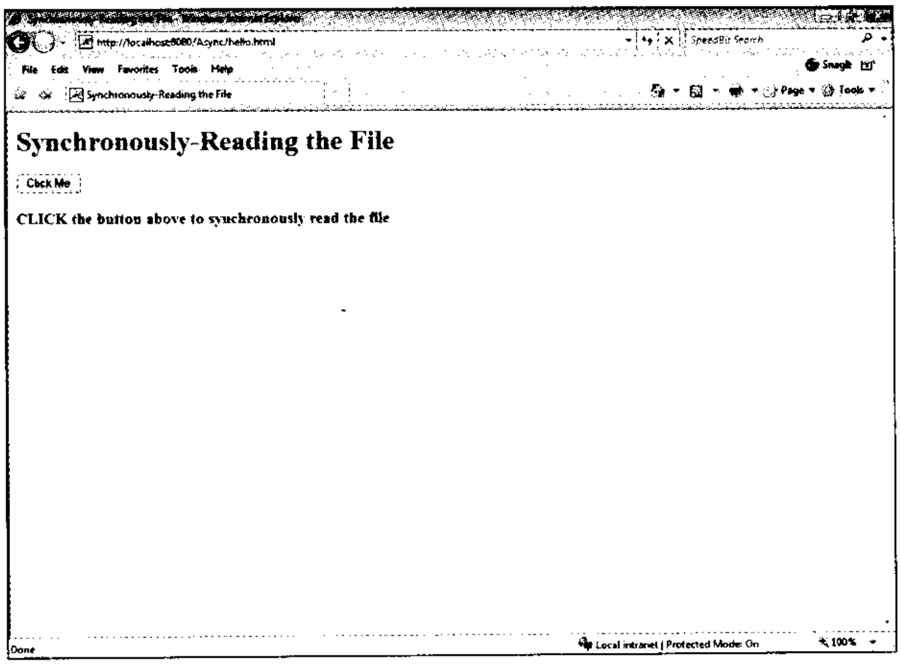
In Listing 30.1, when the Click Me button is clicked the `hello()` method is called. This method, in turn, calls the `getXHR()` function, which returns an `XMLHttpRequest` object depending upon the browser in use. After that the synchronous request is made for `helloAjax.xml` file.

The code given in Listing 30.2 shows code for the `helloAjax.xml file` (you can find this file in the Code/AJAX/Chapter 30/Async folder on the CD):

Listing 30.2: The `helloAjax.xml File`

```
<class>

<student>

<name> Suchita </name>

<subject>Economics</subject>
</student>
<student>
<name> Charu </name>

<subject>Accountancy</subject>

</student>

</class>
```

Until the entire file is loaded, no other operation is performed and the user has to wait. Then, after the response is received, it is displayed on the browser. Figure 30.1 shows how the `hello.html` page looks like:

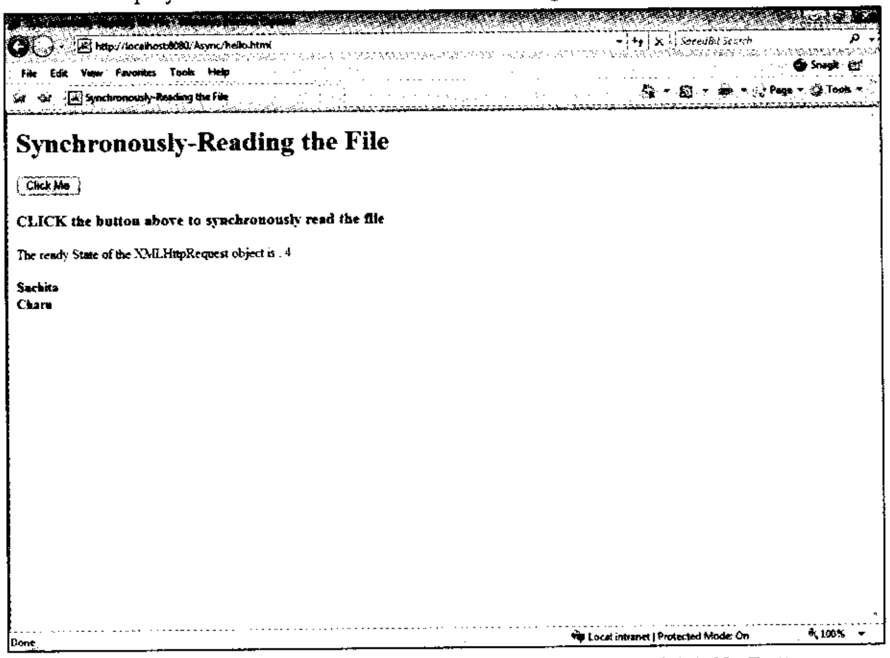

http://localhost:8080/Async/hello.html
File Edit View Favorites Tools Help
Synchronously-Reading the File
Click Me
Done
Local intranet | Protected Mode: On
100%

Synchronously-Reading the File

CLICK the button above to synchronously read the file

Figure 30.1: Displaying the hello.html Page

After clicking the 'Click Me' button, shown in the Figure 30.1, the `helloAjax.xml` file is loaded and the response received is displayed on the browser, as shown in the Figure 30.2:


http://localhost:8080/Async/hello.html
File Edit View Favorites Tools Help
Synchronously-Reading the File
Click Me
Done
Local intranet | Protected Mode: On
100%

Synchronously-Reading the File

CLICK the button above to synchronously read the file

The ready State of the XMLHttpRequest object is . 4

Sachita
Charu

Figure 30.2: Displaying hello.html Page after Clicking the Click Me Button

By now it is clear that during the state change of XMLHttpRequest object, no other operation can be performed by the user, given in the synchronous mode. On the contrary, the asynchronous mode prevents the waiting mode of the users. Therefore, let's see how to read a file asynchronously.

Reading a File Asynchronously

In the following code, when the request is made for helloAjax.xml file, the request is made asynchronously. While the browser waits for the response, the user is aware of the state of XMLHttpRequest object. Moreover, while the response is received, the progressbar.gif image is displayed and after the response is received the response is displayed on the browser.

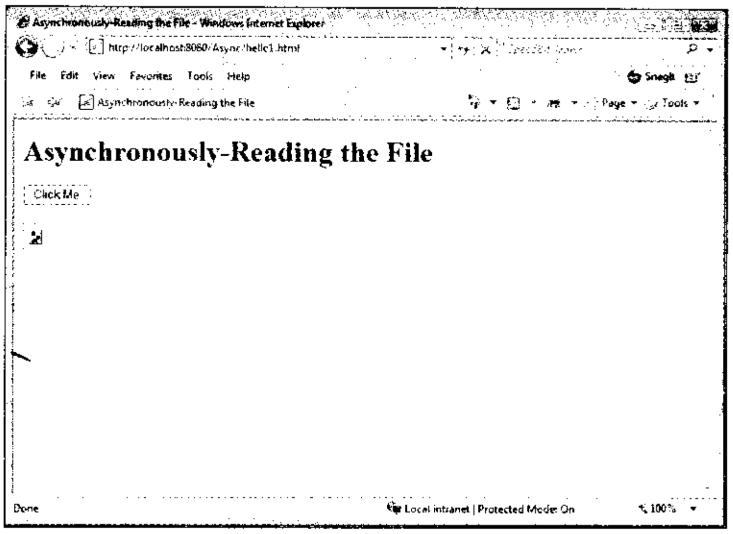
The code given in Listing 30.3 shows code for the hello1.html file, which asynchronously reads helloAjax.xml file (you can find this file in Code/AJAX/Chapter 30/Async folder on the CD):

Listing 30.3: The hello1.html File

```
<html>
<head>
<title>Asynchronously-Reading the File</title>
<script type="text/javascript">
function getXHR()
{
  var RequestObject;
  if (window.XMLHttpRequest) {
        RequestObject = new XMLHttpRequest();
  } else if (window.ActiveXObject)
  {
        RequestObject = new ActiveXObject("Microsoft.XMLHTTP");
  }
  return RequestObject;
}
function hello1() {
  var request = getXHR();
  request.open("Get","helloAjax.xml",true);
  request.onreadystatechange=function() {
        if(request.readyState==4) {
        document.getElementById('img1').src="";
        document.getElementById('img1').alt = "The response has been received.....";
        if(request.status==200) {
                var nodes=request.responseXML.getElementsByTagName("name");
                for (i=0; i<nodes.length; i++) {
                        document.getElementById('message').innerHTML += '<b>'
                         + nodes[i].firstChild.nodeValue + '</b><br/>';
                }
        }
        else {
                alert ("Request failed");
        }
        }
        else {
                document.getElementById('img1').src = "progressbar.gif";
                alert(request.readyState);
        }
  }
  request.send(null);
}
</script>
</head>
<body>
```

```
<h1>Asynchronously-Reading the File</h1>
<input type="button" value="Click Me" onclick="hello1()"/>
<p>
<img id="img1"></img>
</p>
<DIV id="message"></DIV>
</body>
</html>
```

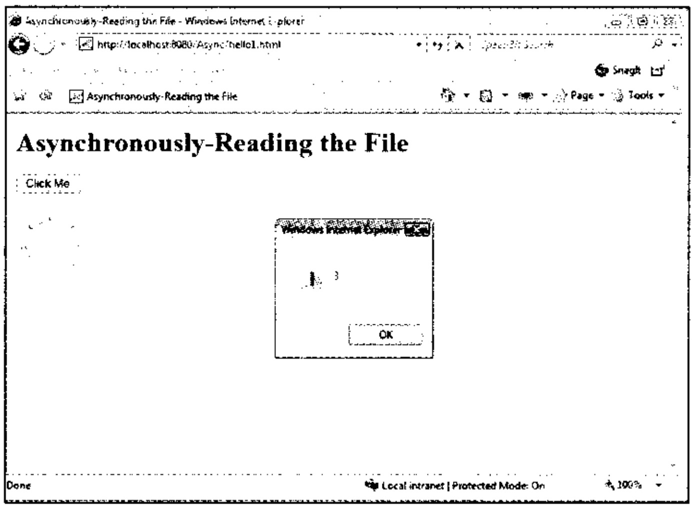
Figure 30.3 displays the `hello1.html` page:


Asynchronously-Reading the File - Windows Internet Explorer
http://localhost:8080/Async/hello1.html
File Edit View Favorites Tools Help
Page Tools
Done
Local intranet | Protected Mode: On
100%

Asynchronously-Reading the File

Figure 30.3: Displaying hello1.html Page

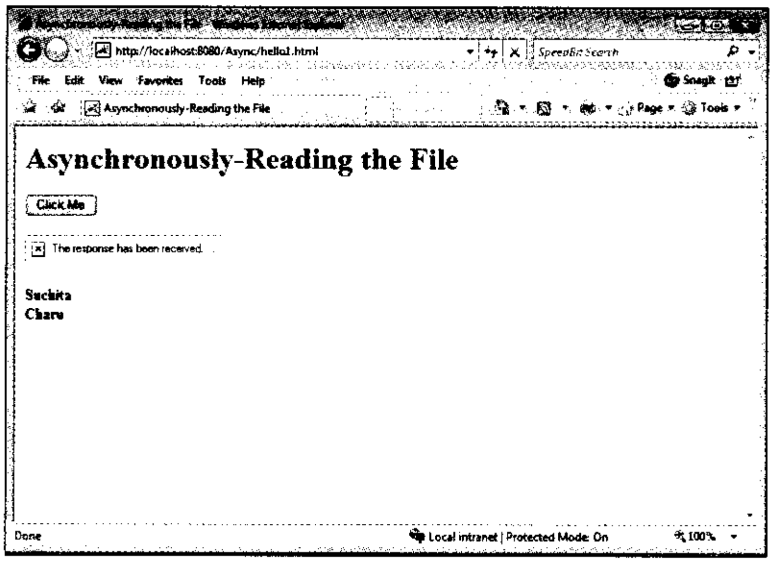
When the user clicks the Click Me button, shown in the Figure 30.3, the `XMLHttpRequest` object is created and the asynchronous request is made for `helloAjax.xml` file. As the state of `XMLHttpRequest` object changes, the image on the browser changes to the `progressbar.gif`, as shown in the Figure 30.4, and the alert box displays the state of `XMLHttpRequest` object:


Asynchronously-Reading the File - Windows Internet Explorer
http://localhost:8080/Async/hello1.html
Page Tools
Windows Internet Explorer
3
OK
Done
Local intranet | Protected Mode: On
100%

Asynchronously-Reading the File

Figure 30.4: Displaying hello1.html Page after the Button Click

Similarly, as shown in Figure 30.4, the alert box displays the state of `XMLHttpRequest` object and when the `readyState` is equal to 4, the response is received and is displayed on the browser, as shown in the Figure 30.5:


http://localhost:8080/Async/hello1.html
File Edit View Favorites Tools Help
Asynchronously-Reading the File
Click Me
Done
Local intranet | Protected Mode: On
100%

Asynchronously-Reading the File

The response has been received.

Suchita
Charu

Figure 30.5: Displaying Response after the Asynchronous Request is Made

By now you must have understood the concept and difference between synchronous and asynchronous request mode. The asynchronous mode of request and response enables the user to perform other operations, instead of waiting for the response.

Let's now discuss some commonly used AJAX techniques, which are implemented using XMLHttpRequest object.

Using Basic AJAX Techniques

AJAX technique is used in various ways—to perform live searches, implement auto-complete, download images, drag and drop, play games, and add interactivity to maps. Now we will discuss some AJAX techniques and their implementations. You can use these techniques in your Web application to make it an enriched one. We have used Java Servlets as server-side component but you can use any server-side technology, like .NET, Ruby, Perl, PHP, etc.

Performing Validation

Web applications must inform the users about errors as soon as possible. Earlier, Web-based applications used to submit the whole page to validate the data or would depend on JavaScript to check the form. However, some operations cannot be performed in JavaScript. The same validation logic has to be repeated on both client-side and server-side, since it is possible that JavaScript is not enabled on the user's browser. With Ajax, you can simply invoke validation function written for server.

The code given in Listing 30.4 illustrates the user interface of Validation (you can find this file in Code/AJAX/Chapter 30/Validation folder on the CD):

Listing 30.4: The `validation.html` page of Validation Application

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" >
<html>
<head>
<title>Using Ajax for validation</title>
<script type="text/javascript">
var xmlHttp;
function createXMLHttpRequest() {
    if (window.ActiveXObject) {
          xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else if (window.XMLHttpRequest) {
          xmlHttp = new XMLHttpRequest();
    }
 }
 function validate() {
    createXMLHttpRequest();
    var number = document.getElementById("number");
    var url = "ValidationServlet?number=" + escape(number.value);
    xmlHttp.open("GET", url, true);
    xmlHttp.onreadystatechange = callback;
    xmlHttp.send(null);
 }
 function callback() {
    if (xmlHttp.readyState == 4) {
          if (xmlHttp.status == 200) {
                var mes =
                xmlHttp.responseXML
                .getElementsByTagName("message")[0].firstChild.data;
                var val =
                xmlHttp.responseXML
                .getElementsByTagName("passed")[0].firstChild.data;
                setMessage(mes, val);
          }
    }
 }
 function setMessage(message, isValid) {
    var messageArea = document.getElementById("numMessage");
    var fontColor="red";
    if (isValid == "true") {
          fontColor = "green";
    }
 messageArea.innerHTML = "<font color=" + fontColor + ">"+ message + " </font>";
 }
 </script>
 </head>
 <body>
 <h1>Ajax Validation Example</h1>
 Number: <input type="text" size="10" id="number" onchange="validate();"/>
 <div id=numMessage>
 </body>
 </html>
```

Listing 30.4 has the main validate method invoked on onchange event trigger, which in turn makes a call to `create XMLHttpRequest` method to create the XHR object depending upon the browser. The `open()` method used here specifies that this is a `GET` request and makes the endpoint URL, which in this case, contains the encoded parameters. Then the `send()` method sends the request to the server. The `onreadystatechange` property stores the pointer to callback function which is called whenever the `XMLHttpRequest` object's internal state changes. The callback function is also responsible for calling the specified function to handle the response whenever the response reaches. The callback function gets the results from the server and is then passed to the `setMessage` method, which determines the color in which the message should be displayed. After the JavaScript part, the page has one text field number which is displayed to the user. The `div numMessage` tag is used to reference itself in JavaScript.

The `ValidationServlet` fetches request parameter "number" and calls the `validateInteger` function on that parameter. The `validateInteger` function parses the String "number" and tries to convert it into integer. If the conversion occurs, it returns `true`, otherwise it returns `false`. The return value is stored in a boolean variable, which is "passed". Depending on this boolean value, the string message is set and this message is sent as XML response.

The code given in Listing 30.5 shows code for the ValidationServlet.java file (you can find this file in Code/AJAX/Chapter 30/Validation/src folder on the CD):

Listing 30.5: The ValidationServlet.java file of Validation Application

```
import java.io.*;
import java.lang.NumberFormatException;
import javax.servlet.*;
import javax.servlet.http.*;
public class ValidationServlet extends HttpServlet {
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  PrintWriter out = response.getWriter();
  boolean passed = validateInteger(request.getParameter("number"));
  response.setContentType("text/xml");
  response.setHeader("Cache-Control", "no-cache");
  String message = "You have entered an invalid number.";
  if (passed) {
              message = "You have entered a valid number.";
  }
  out.println("<response>");
  out.println("<passed>" + Boolean.toString(passed) + "</passed>");
  out.println("<message>" + message + "</message>");
  out.println("</response>");
  out.close();
}
private boolean validateInteger(String number) {
  boolean isValid = true;
  if( number!= null) {
        try {
              Integer.parseInt(number);
        }
        catch(NumberFormatException e) {
              isValid = false;
        }
  }
  else
        isValid = false;
        return isValid;
  }
}
```

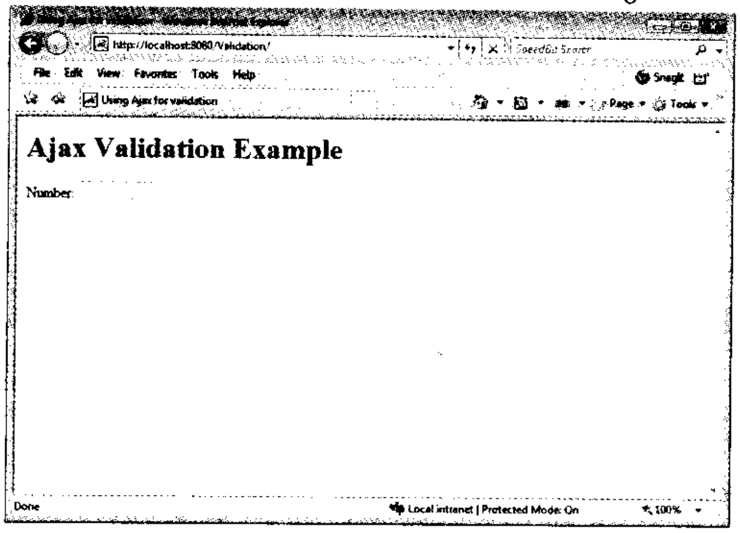
For mapping ValidationServlet in Listing 30.5, we need to write the web.xml. The code given in Listing 30.6 maps the validation.html home page as welcome file in web.xml (you can find this file in Code/AJAX/Chapter 30/Validation/WEB-INF folder on the CD):

Listing 30.6: The web.xml File of the Validation Application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<servlet>
<servlet-name>ValidationServlet</servlet-name>
<servlet-class>ValidationServlet</servlet-class>
</servlet>
<servlet-mapping>
```

```
<servlet-name>ValidationServlet</servlet-name>
<url-pattern>/ValidationServlet</url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file>validation.html</welcome-file>
</welcome-file-list>
</web-app>
```

Make a new Web Project and name it Validation. Then compile `ValidationServlet.java` and properly package these files in root directory "Validation" of this Web application. Make sure the /WEB-INF/lib folder has the essential Servlet runtime jar files. Deploy the application on Tomcat 6 version, open Internet Explorer, and type `http://localhost:8080/Validation/`. The result is shown in Figure 30.6:

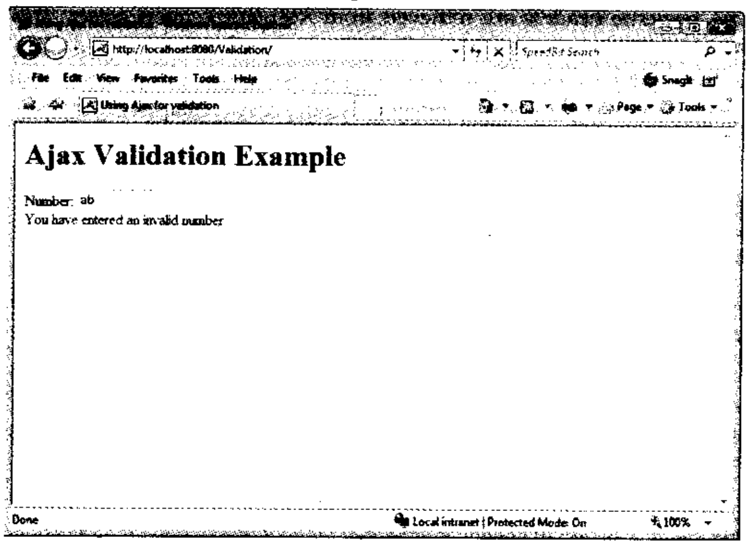

http://localhost:8080/Validation/
File Edit View Favorites Tools Help
Using Ajax for validation
Done
Local intranet | Protected Mode: On
100%

Ajax Validation Example

Number:

Figure 30.6: Home Page for Validation Application

In Figure 30.6, type `ab` text string in the Number textbox and click outside the textbox. A message "You have entered an invalid number" appears, as shown in Figure 30.7:

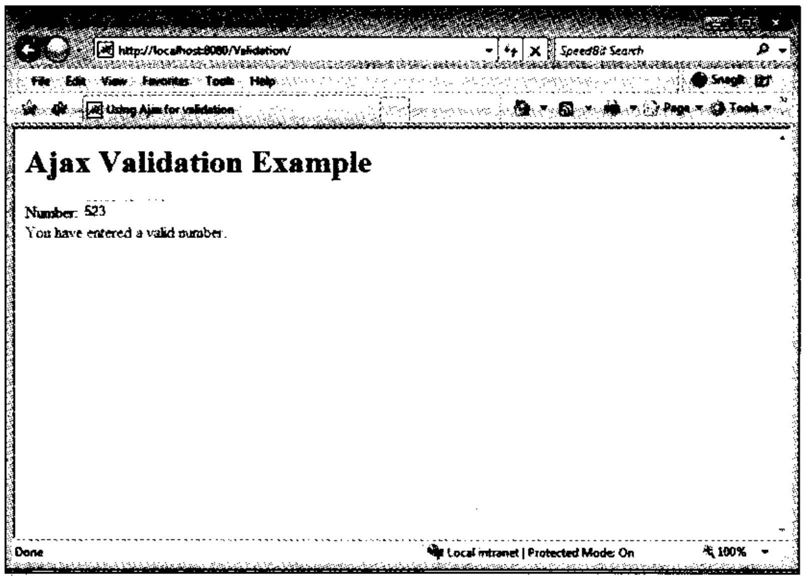

http://localhost:8080/Validation/
File Edit View Favorites Tools Help
Using Ajax for validation
Done
Local intranet | Protected Mode: On
100%

Ajax Validation Example

Number: ab
You have entered an invalid number

Figure 30.7: Checking Invalid Value

Again enter a value, say 523, in the 'Number' textbox. You now receive a message "You have entered a valid number", as shown in the Figure 30.8:



Ajax Validation Example

Number: 523

You have entered a valid number.

Figure 30.8: Checking Valid Value

The data entered in the number textbox is validated and respective result is shown on the page, as shown in Figure 30.8. In this section, you have performed validation of data using the validation technique of AJAX programming; let's discuss next technique of AJAX programming in the following section.

Reading Response Headers

Sometimes you are not interested to get content from the server, but only want to access the information related to the content, such as type of content, its length, and modification date. The response headers include Content-Type, Content-Length and Last-Modified date, etc. Here, the main idea is to send HEAD request to the server. When a server responds to the HEAD request, it sends only the response headers without the content. Listing 30.7 illustrates how response headers are retrieved from XMLHttpRequest object. This page has one link to execute method on XMLHttpRequest object to read all the response headers.

The code given in Listing 30.7 shows how response headers are retrieved from XMLHttpRequest object (you can find the `readingResponseHeaders.html` file in the Code/AJAX/Chapter 30/ResponseHeaders folder on the CD):

Listing 30.7: The `readingResponseHeaders.html` page of `ResponseHeaders` Application

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Reading Response Headers</title>
<script type="text/javascript">
var xmlHttp;
var requestType = "";
function createXMLHttpRequest() {
  if (window.ActiveXObject) {
       xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else if (window.XMLHttpRequest) {
       xmlHttp = new XMLHttpRequest();
```

```
      }
   }
   function doHeadRequest(request, url) {
     requestType = request;
     createXMLHttpRequest();
     xmlHttp.onreadystatechange = handleStateChange;
     xmlHttp.open("HEAD", url, true);
     xmlHttp.send(null);
   }
   function handleStateChange() {
     if(xmlHttp.readyState == 4) {
           if(requestType == "allResponseHeaders") {
                  getAllResponseHeaders();
           }

     }
   }
   function getAllResponseHeaders() {
     alert(xmlHttp.getAllResponseHeaders());
   }

   </script>
   </head>
   <body>
   <h1>Reading Response Headers</h1>
   <a href="javascript:doHeadRequest('allResponseHeaders','
    readingResponseHeaders.xml');">
   Read All Response Headers</a>
   <br/>
   </body>
   </html>
```

When a user clicks on 'Read All Response Headers' hyperlink, as shown in Figure 30.9, the request is passed to doHeadRequest method with request type allResponseHeaders and url readingResponseHeaders.xml. This method calls the createXMLHttpRequest method, which, in turn, invokes the handleStateChange method. When the server response is complete, the handleStateChange method is called. The handleStateChange method checks if the responseType is allResponseHeaders and then calls the user-defined getAllResponseHeaders, which, in turn, calls the default getAllResponseHeaders method of xmlHttp.

The code given in Listing 30.8shows an empty XML file used to generate more response headers in Figure 30.9:

Listing 30.8: The readingResponseHeaders.xml file of ResponseHeaders Application.

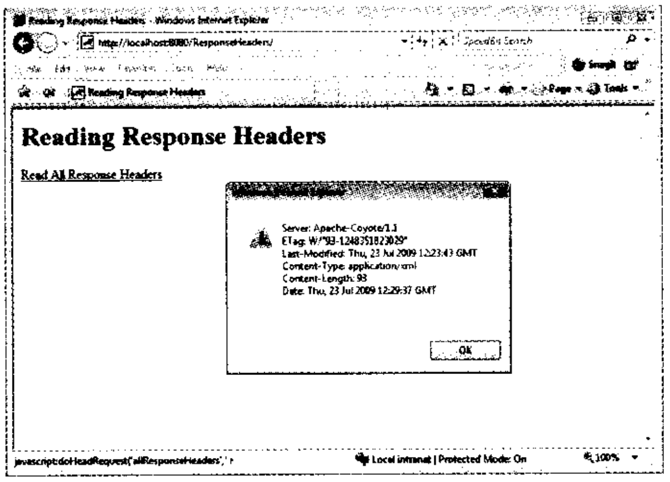
```
<?xml version="1.0" encoding="UTF-8"?>
<readingResponseHeaders>
</readingResponseHeaders>
```

Now make a new Web Project and name it ResponseHeaders. Properly package files in Listing 30.7 and Listing 30.8 in the root directory of the ResponseHeaders Web application. The `readingResponseHeaders.html` file is included as welcome file in `web.xml`. Deploy the application on Tomcat 6 version.

NOTE

Edit the web.xml file of listing 30.6 to map the `readingResponseHeaders.html` file as the welcome file of the application.

Open Internet Explorer and type `http://localhost:8080/ResponseHeaders/`. Figure 30.9 containing "`Read All Response Headers`" hyperlink appears. When the user clicks on this link, the alert dialog box, as shown in Figure 30.9 containing all response headers, is displayed:


Reading Response Headers - Windows Internet Explorer
http://localhost:8080/ResponseHeaders/
Reading Response Headers
Read All Response Headers
OK
javascript:doHeadRequest('allResponseHeaders','
Local intranet | Protected Mode: On
100%

```
Server: Apache-Coyote/1.1
ETag: W/"93-1248351823029"
Last-Modified: Thu, 23 Jul 2009 12:23:43 GMT
Content-Type: application/xml
Content-Length: 93
Date: Thu, 23 Jul 2009 12:29:37 GMT
```

Figure 30.9: Displaying all Response Headers

You have learned handling response headers in an AJAX application, in this section. Let's discuss another AJAX technique using which you can load list boxes dynamically, in the following section.

Loading Dynamically List Boxes

Web applications are often made in step by step manner, which means each page asks the user for small pieces of input and the succeeding page's data is built from previous page's input. Earlier, it was difficult to update a page dynamically, especially in cases where some fields of page needed to change based on user input, without refreshing the entire page. But with AJAX, all this is possible with ease.

Suppose there are two select boxes, X and Y, in an application. In cases where values of select box Y are filled depending on the selected value in select box X, the available values for select box Y can be held in hidden select boxes. When the selected value in select box X changes, JavaScript can predict which hidden select should be displayed. That select box can then be made visible and the previous select box can be hidden. Another way is to dynamically populate the option elements of select box B with the elements from a hidden list box. We are using the second technique.

The application under this section populates subcategories depending upon the selected category. There are three categories in the first select box—PL (Programming Languages), Furniture and Clothes, respectively. Each of these categories has three sub-categories. PL has C, C++, and Java sub-categories; Furniture has Table, Chair, and Bed sub-categories; and Clothes has Cotton, Nylon and Woolen sub-categories. There are total nine combinations made from these categories. If there are more combinations, then it becomes difficult to populate them with JavaScript alone.

You can solve this problem easily using Ajax techniques. Each time a selection in the category select box changes, an asynchronous request is sent to the server requesting a list of sub-categories available for that particular category. The server is responsible for determining the list of sub-categories for the category requested by the browser. You can also fetch these values from database. But for simplicity, we hard coded them. Once the available sub-categories are found, the server packages them in an XML file and returns them to the browser.

The browser is responsible for parsing the server's XML response and populating the `subcats` element.

The code given in Listing 30.9 shows code for the `dynamicLists.html` page which illustrates the dynamically creation of the contents of one select box based on the value of other list boxes (you can find this file in the Code/AJAX/Chapter 30/DynamicLoadListBoxes folder on the CD):

Listing 30.9: The `dynamicLists.html File`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
<title>Dynamically Filling Lists</title>
<script type="text/javascript">
  var xmlHttp;
  function createXMLHttpRequest() {
        if (window.ActiveXObject) {
              xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
        }
        else if (window.XMLHttpRequest) {
              xmlHttp = new XMLHttpRequest();
        }
  }
        function refreshCatList() {
        var cat = document.getElementById("cat").value;
        if(cat == "") {
              clearCatsList();
              return;
        }
        var url = "RefreshCatListServlet?"+ createQueryString(cat);
        createXMLHttpRequest();
        xmlHttp.onreadystatechange = handleStateChange;
        xmlHttp.open("GET", url, true);
        xmlHttp.send(null);
  }
  function createQueryString(cat) {
        var queryString = "cat=" + cat;
        return queryString;
  }
  function handleStateChange() {
        if(xmlHttp.readyState == 4) {
              if(xmlHttp.status == 200) {
                    updateCatsList();
              }
        }
  }
  function updateCatsList() {
        clearCatsList();
        var subcats = document.getElementById("subcats");
        var results = xmlHttp.responseXML.getElementsByTagName("subcat");
        var option = null;
        for(var i = 0; i < results.length; i++) {
              option = document.createElement("option");
              option.appendChild(document.createTextNode(results[i].
               firstChild.nodeValue));
              subcats.appendChild(option); }
  }
  function clearCatsList() {
        var subcats = document.getElementById("subcats");
        while(subcats.childNodes.length > 0) {
              subcats.removeChild(subcats.childNodes[0]); }
  }
  </script>
</head>
<body>
<h1>Select Category</h1>
<br/><br/>
<span style="font-weight:bold;">Category:</span>
<select id="cat" onchange="refreshCatList();">
<option value="">Select One</option>
```

```
<option value="PL">PL</option>
<option value="Furniture">Furniture</option>
<option value="Clothes">Clothes</option>
</select>
<br/><br/>
<span style="font-weight:bold;">Subcats:</span>
<br/>
<select id="subcats" size="6" style="width: 300px;">
</select>
</form>
</body>
</html>
```

When we select one of the options of Category List box, the `onchange` event triggers a call to `refershCatList()` function. This method gets reference to the select box with `cat id`. If no value is selected, then the `clearCatsList` is invoked. Then the url to invoke RefreshCatListServlet Servlet is made using `createQueryString()` method. This Servlet is responsible for handling the request. After creating the `XMLHttpRequestObject`, request to `RefreshCatListServlet` is made. The `onreadystatechange` property invokes the `handlestatechange ()` method which has code for correctly received response. At last, this function makes a call to the `updateCatsList()` method. The `updateCatsList()` method gets subcats head element from XML response and retrieves all its nested `<subcat >` elements. Then, it makes option elements one by one and appends each as a child of `var` subcats.

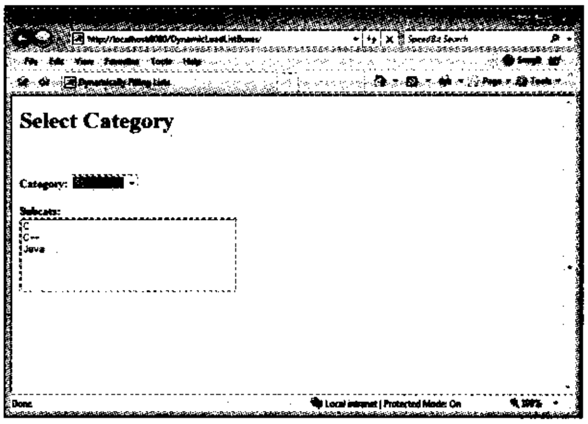
The code given in Listing 30.10 makes XML response for DynamicLoadListBoxes application (you can find the `RefreshCatListServlet.java` file in the Code/AJAX/Chapter 30/DynamicLoadListBoxes/src folder on the CD):

Listing 30.10: The `RefreshCatListServlet.java` file of `DynamicLoadListBoxes` Application

```
import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;
public class RefreshCatListServlet extends HttpServlet
{
  private static ArrayList availableCats = new ArrayList();
  protected void processRequest(HttpServletRequest request
   , HttpServletResponse response)
  throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        String cat = request.getParameter("cat");
        StringBuffer results = new StringBuffer("<subcats>");
        MakeCategory availableCategory=null;
        for(Iterator it = availableCats.iterator(); it.hasNext();) {
                availableCategory = (MakeCategory)it.next();
                if(availableCategory.cat.equals(cat)) {
                        results.append("<subcat>");
                        results.append(availableCategory.subcat);
                        results.append("</subcat>");
                }
        }
  results.append("</subcats>");
  response.setContentType("text/xml");
  response.getWriter().write(results.toString());
  }
  protected void doGet(HttpServletRequest request,
   HttpServletResponse response)
  throws ServletException, IOException {
        processRequest(request, response);
    }
    public void init() throws ServletException {
        availableCats.add(new MakeCategory("PL", "C"));
        availableCats.add(new MakeCategory("PL", "C++"));
        availableCats.add(new MakeCategory("PL", "Java"));
        availableCats.add(new MakeCategory("Furniture", "Table"));
        availableCats.add(new MakeCategory("Furniture", "Chair"));
        availableCats.add(new MakeCategory("Furniture", "Bed"));
        availableCats.add(new MakeCategory("Clothes", "Cotton"));
        availableCats.add(new MakeCategory("Clothes", "Nylon"));
        availableCats.add(new MakeCategory("Clothes", "Woolen"));
    }
    private static class MakeCategory {
        private String cat;
        private String subcat;
        public MakeCategory(String cat, String subcat) {
            this.cat = cat;
            this.subcat = subcat;
        }
    }
}
```

In Listing 30.10, the first static variable, `availableCats` variable, of `ArrayList` type is created. Then the `init()` function of `RefreshCatListServlet` Servlet is called, which adds variables of `MakeCategory` type to availableCats one by one. Objects of `MakeCategory` class are made with the help of `MakeCategory` parameterized constructor. Now the `doGet()` method of Servlet is called which, in turn, calls the `processRequest()` method. Here, the Servlet first fetches the request parameter cat. Once the requested category is determined, the Servlet iterates over a collection of objects representing the available category and subcategory combinations. If a particular object's category matches the requested category, then the object's subcategory property is added to the response XML string. Once all the subcategories for the specified category have been found, the response XML is written back to the browser.

Now let's make a new Web Project and name it `DynamicLoadListBoxes`. Compile `RefreshCatListServlet.java` and properly package files in Listing 30.9 and Listing 30.10 in root directory `DynamicLoadingListBoxes` of Web application. We have not shown `web.xml` here, but you have to create this file having mapping of Servlet. The `dynamicLists.html` is included as welcome file in `web.xml` Deploy the application on Tomcat 6 version. Open Internet Explorer browser and type `http://localhost:8080/DynamicLoadListBoxes/`. In Figure 30.10, when you select the PL category, the list box containing C, C++, and Java sub-categories is displayed:


Select Category

Category: PL

Subcats:

C
C++
Java

Figure 30.10: Showing Loading of List Box Dynamically

In this section, you have learned loading a list box dynamically, in an AJAX application. Let's discuss creating an AJAX application with auto refresh feature, in the following section.

Auto Refreshing Page

We all know that there are pieces of information which change frequently, like headline news, weather data on web sites (www.news.google.com, www.weather.com), etc. In that case, it is not worthwhile to repaint the entire page when the changes in only one or two headlines are needed. Also after the entire page refreshes, we feel difficult to find what is new. The following code, Listing 30.11, automatically updates itself by one message after 5 seconds. The dynamicUpdates.html page has one simple button "Start" clicking which starts the AutoRefreshing process.

The code given in Listing 30.11 shows code for the dynamicUpdates.html page (you can find the `dynamicUpdates.html` file in the Code/AJAX/Chapter 30/AutoRefreshing folder on the CD):

Listing 30.11: The `dynamicUpdates.html` document of `AutoRefreshing` Application

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Ajax Dynamic Update</title>
<script type="text/javascript">
var xmlHttp;
function createXMLHttpRequest() {
  if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
  }
}
function doStart() {
  createXMLHttpRequest();
  var url = "DynamicRefreshServlet?function=reset";
  xmlHttp.open("GET", url, true);
  xmlHttp.onreadystatechange = startCallback;
  xmlHttp.send(null);
}
function startCallback() {
  if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
                setTimeout("pollingServer()", 5000);
                refreshTime();
        }
  }
}
function pollingServer() {
  createXMLHttpRequest();
  var url = "DynamicRefreshServlet?function=continue";
  xmlHttp.open("GET", url, true);
  xmlHttp.onreadystatechange = pollCallback;
  xmlHttp.send(null);
}
function refreshTime() {
  var time_span = document.getElementById("time");
  var time_val = time_span.innerHTML;
  var int_val = parseInt(time_val);
  var new_int_val = int_val - 1;
  if (new_int_val > -1) {
        setTimeout("refreshTime()", 1000);
```

```
            time_span.innerHTML = new_int_val;
        } else {
            time_span.innerHTML = 5;
        }
    }
    function pollCallback() {
        if (xmlHttp.readyState == 4) {
            if (xmlHttp.status == 200) {
                var message =
                xmlHttp.responseXML
                .getElementsByTagName("message")[0].firstChild.data;
                if (message != "done") {
                var new_row = createRow(message);
                var table = document.getElementById("dynamicRefreshArea");
                var table_body =
                table.getElementsByTagName("tbody").item(0);
                var first_row =
                table_body.getElementsByTagName("tr").item(1);
                table_body.insertBefore(new_row, first_row);
                setTimeout("pollingServer()", 5000);
                refreshTime();
                }
            }
        }
    }
    function createRow(message) {
        var row = document.createElement("tr");
        var cell = document.createElement("td");
        var cell_data = document.createTextNode(message);
        cell.appendChild(cell_data);
        row.appendChild(cell);
        return row;
    }
    </script>
    </head>
    <body>
    <h1>Ajax Autorefeshing Example</h1>
    This page will automatically refresh itself:
    <input type="button" value="Start" id="go" onclick="doStart();"/>
    <p>
    Page will refresh in <span id="time">5</span> seconds.
    <p>
    <table id="dynamicRefreshArea" align="left">
    <tbody>
    <tr id="row0"><td></td></tr>
    </tbody>
    </table>
    </body>
    </html>
```

In Listing 30.11, the `onclick` event triggers a call to `doStart()` method. It first creates the `XMLHttpRequest` object, and then the request url to invoke the DynamicRefreshServlet is made with the function =reset query string. The `onreadystatechange` property invokes the `startCallback()` method. The `startCallback()` method has the `setTimeout()` method, which invokes the `pollingServer()` method after 5 seconds. In between these 5 seconds, the `refreshTime()` method is invoked. The `refreshTime()` is a recursive function and changes the time value on user interface from 5 to 0. It is invoked after 1 sec. In the refreshTime method, when the `new_int_val` reaches `-1`, its `innerHTML` property is again set to 5. The `pollingServer()` method again creates another instance of `XMLHttpRequest` object. Now the request URL is made with

"DynamicRefreshServlet?function=continue" query string. On Servlet side, the message string is initialized and the XML response is made with <message> tags. Coming back to dynamicUpdates.html file, the `onreadystatechange` property is initialized to `pollCallback()` method. The `pollCallback()` method fetches the message from message tag of XML response and stores it in variable message. If the message is not equal to the last message "finish", it calls the `createRow()` method. This function creates a row of table from the current message and adds it to the table element having id equal to dynamicRefreshArea. The first message is displayed as a row of table. At the end of `pollCallback()` method, calls to `pollingServer()` and `refreshTime()` methods are made again. Finally, all the messages are displayed in row form of the table as shown in Figure 30.11. The dynamicUpdates.html page accesses all the messages from the `DynamicRefreshServlet`, which is given in Listing 30.12.

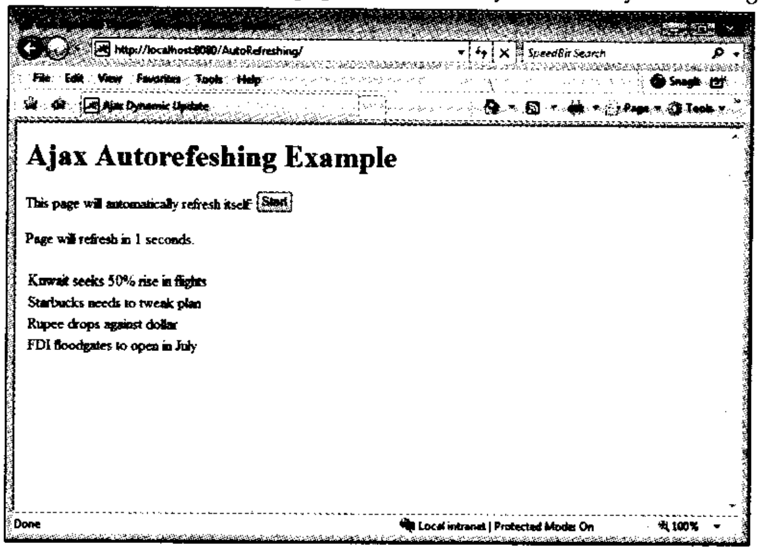
The code given in Listing 30.12 returns one message based on simple counter (you can find the `DynamicRefreshServlet.java` file in the Code/AJAX/Chapter 30/AutoRefreshing/src folder on the CD):

Listing 30.12: The `DynamicRefreshServlet.java` file of AutoRefreshing application

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class DynamicRefreshServlet extends HttpServlet {
private int count = 1;
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  String res = "";
  String function = request.getParameter("function");
  String message = "";
  if (function.equals("reset")) {
        count = 1;
  } else {
  switch (count) {
        case 1: message = "FDI floodgates to open in July "; break;
        case 2: message = "Rupee drops against dollar "; break;
        case 3: message = "Starbucks needs to tweak plan "; break;
        case 4: message = "Kuwait seeks 50% rise in flights "; break;
        case 5: message = "IOC plans to produce biodiesel "; break;
        case 6: message = "Banks largest spender on IT : Nasscom"; break;
        case 7: message = "finish"; break;
  }
  count++;
  }
  res = "<message>" + message + "</message>";
  PrintWriter out = response.getWriter();
  response.setContentType("text/xml");
  response.setHeader("Cache-Control", "no-cache");
  out.println("<response>");
  out.println(res);
  out.println("</response>");
  out.close();
}
}
```

This file has the count variable, which tells the number of messages. Then comes the Servlet request processing `doGet()` method. The String variable `res` is the XML response, which is initially initialized to empty string. The Servlet first fetches the "function" parameter and, depending upon its value which is either reset or continue, the value of count is set. For each message to be displayed on the screen, a call to the `DynamicRefreshServlet` is made.

Make a new Web Project and name it AutoRefreshing. Compile `DynamicRefreshServlet.java` and properly package files in Listing 30.11 and Listing 30.12 in root directory of the AutoRefreshing Web application. The dynamicUpdates.html is included as welcome file in web.xml. Deploy the application on Tomcat 6 version. Open Internet Explorer and type http://localhost:8080/AutoRefreshing/. In Figure 30.11 that appears, when the user clicks on the 'Start' button, the page is automatically refreshed by one message in 5 seconds:


http://localhost:8080/AutoRefreshing/
SpeedBit Search
File Edit View Favorites Tools Help
Ajax Dynamic Update
Page Tools
Done
Local intranet | Protected Mode: On
100%

Ajax Autorefeshing Example

This page will automatically refresh itself Start

Page will refresh in 1 seconds.

Kuwait seeks 50% rise in flights
Starbucks needs to tweak plan
Rupee drops against dollar
FDI floodgates to open in July

Figure 30.11: Showing Autorefreshing of Page

In this section, you have learned creating an AJAX application that refresh itself automatically in 5 seconds, as shown in Figure 30.11. Now, let's discuss another AJAX technique in the following section.

Dynamic Progress Bar

Almost all Web applications need to call long running processes, like adding an item to shopping cart, searching for a profile, etc. from time to time. There should be some way to see the status of those processes on the page of the Web application. Progression bar is used in long running processes to give you an idea about how much the process is completed. After building a progression bar, you can attach it to the Web application.

The code given in Listing 30.13 displays a simple progress bar interface (you can find the ProgressionBar.htmlfile in the Code/AJAX/Chapter 30/ProgressionBar folder on the CD):

Listing 30.13: The ProgressionBar.html code of ProgressionBar application

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Ajax Progress Bar</title>
 <script type="text/javascript">
var xmlHttp;
var key;
var bar_color = 'green';
var span_id = "rectangle";
var clear = "&nbsp;&nbsp;&nbsp;"
function createXMLHttpRequest() {
  if (window.ActiveXObject) {
       xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else if (window.XMLHttpRequest) {
       xmlHttp = new XMLHttpRequest();
  }
}
function proceed() {
    createXMLHttpRequest();
    checkDiv();
    var url = "ProgressionBarServlet?function=make";
    var button = document.getElementById("proceed");
    button.disabled = true;
    xmlHttp.open("GET", url, true);
    xmlHttp.onreadystatechange = proceedCallback;
    xmlHttp.send(null);
  }
  function proceedCallback() {
    if (xmlHttp.readyState == 4) {
          if (xmlHttp.status == 200) {
                setTimeout("pollingServer()", 2000);
          }
    }
  }
  function pollingServer() {
    createXMLHttpRequest();
    var url = "ProgressionBarServlet?function=poll&key=" + key;
    xmlHttp.open("GET", url, true);
    xmlHttp.onreadystatechange = pollingCallback;
    xmlHttp.send(null);
  }
  function pollingCallback() {
    if (xmlHttp.readyState == 4) {
          if (xmlHttp.status == 200) {
                var percent_complete =
                xmlHttp.responseXML
                .getElementsByTagName("percentage")[0].firstChild.data;
                var index = processResult(percent_complete);
                for (var i = 1; i <= index; i++) {
                var elem = document.getElementById("rectangle" + i);
                elem.innerHTML = clear;
                elem.style.backgroundColor = bar_color;
                var next_cell = i + 1;
                if (next_cell > index && next_cell <= 9) {
                document.getElementById("rectangle" + next_cell)
                .innerHTML =
                percent_complete + "%";
                }
                }
                if (index < 9) {
                setTimeout("pollingServer()", 2000);
                } else {
                document.getElementById("completed").innerHTML = "Completed!";
                document.getElementById("proceed").disabled = false;
                }
          }
    }
  }
  function processResult(percent_complete) {
    var ind;
    if (percent_complete.length == 1) {
          ind = 1;
    } else if (percent_complete.length == 2) {
          ind = percent_complete.substring(0, 1);
    } else {
          ind = 9;
    }
    return ind;
  }

  function checkDiv() {
    var progress_bar = document.getElementById("progressionBar");
    if (progress_bar.style.visibility == "visible") {
        clearBar();
        document.getElementById("completed").innerHTML = "";
    } else {
        progress_bar.style.visibility = "visible"
    }
  }
  function clearBar() {
    for (var i = 1; i < 10; i++) {
        var elem = document.getElementById("rectangle" + i);
        elem.innerHTML = clear;
        elem.style.backgroundColor = "white";
    }
  }
  </script>
  </head>
  <body>
  <h1>Ajax Progression Bar</h1>
  Start a long running process:
  <input type="button" value="Start" id="proceed" onclick="proceed()"; />
  <p>
  <table align="center">
  <tbody>
  <tr><td>
  <div id="progressionBar"
  style="padding:2px;border:solid black 2px;visibility:hidden">
  <span id="rectangle1">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle2">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle3">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle4">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle5">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle6">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle7">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle8">&nbsp;&nbsp;&nbsp;</span>
  <span id="rectangle9">&nbsp;&nbsp;&nbsp;</span>
  </div>
  </td></tr>
  <tr><td align="center" id="completed"></td></tr>
  </tbody>
  </table>
  </body>
  </html>
```

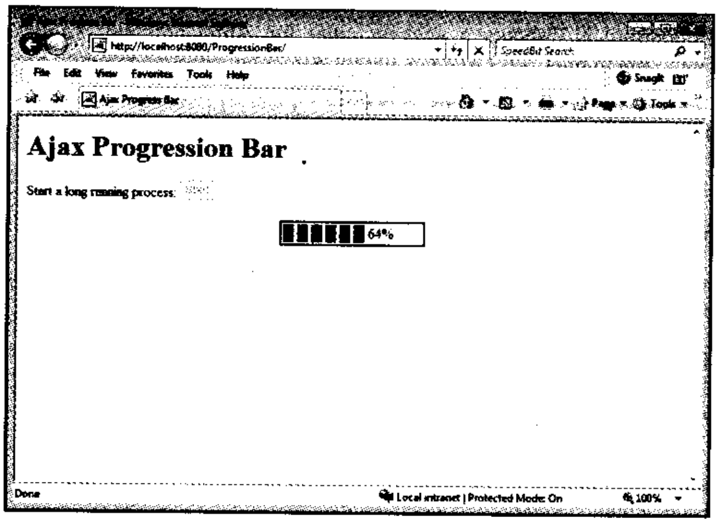
Once a user clicks on Start button, it becomes disabled till the whole progress bar is displayed, as shown in Figure 30.12. The `onclick` event triggers a call to the `proceed()` function, which in turn makes a call to the `checkDiv()` function. The `checkDiv()` function gets reference to the `div` element with id equal to `progressionBar`. Visibility of `progressionBar` is initially set to 'hidden' in style part of this html file. The `checkDiv()` makes it visible. Then the request url to invoke ProgressionBarServlet, as shown in Listing 30.14, is made. The first call to ProgressionBarServlet is made with the query string "function=make". The onreadystatechange property is set to proceedCallback() method. This method calls the pollingServer() method after 2 seconds. In this method, another XHR request object is created. Another call to ProgressionBarServlet is made with query string "function=poll &key=undefined". In pollingServer method, the onreadystatechange property is set to `pollingCallback()` method. The pollingCallback method extracts the percentage value from <percentage> tags of XML response made by ProgressionBarServlet. This method has code to display a rectangular block of width equal to 3 blank space characters with green color and writes percentage value along with the recently displayed rectangular block, as shown in Figure 30.12. The remaining processResult () method just looks for the first digit of the percent completed and on the basis of that digit, it figures out which blocks need to be colored in the progress bar area. For the whole progress bar to be displayed, a total of eight calls to `pollingServer()` are made, since there are eight percentage values in Servlet till index is less than equal to 9. Each invocation takes place after 2 seconds, which means each rectangular block is displayed after two seconds. When the whole bar is displayed, the "Completed!" message appears and the Start button becomes enabled (Figure 30.13). The second time, a user clicks on the Start button, the `clearBar()` method is invoked to clear the bar.

The code given in Listing 30.14 mimics long running process (you can find the `ProgressionBarServlet.java` file in the Code/AJAX/Chapter 30/ProgressionBar/src folder on the CD):

Listing 30.14: The `ProgressionBarServlet.java` file of ProgressionBar Application

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ProgressionBarServlet extends HttpServlet {
private int counter = 1;
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
   String function = request.getParameter("function");
   String res = "";
   if (function.equals("make")) {
         res = "<key>1</key>";
         counter = 1;
   }
   else {
         String percentage = "";
         switch (counter) {
                 case 1: percentage = "10"; break;
                 case 2: percentage = "23"; break;
                 case 3: percentage = "35"; break;
                 case 4: percentage = "51"; break;
                 case 5: percentage= "64"; break;
                 case 6: percentage = "73"; break;
                 case 7: percentage= "89"; break;
                 case 8: percentage = "100";break;
         }
   counter++;
   res = "<percentage>" + percentage + "</percentage>";
   }
   PrintWriter out = response.getWriter();
   response.setContentType("text/xml");
   response.setHeader("Cache-Control", "no-cache");
   out.println("<response>");
   out.println(res);
   out.println("</response>");
   out.close();
 }
 }
```

Make a new Web Project and name it ProgressionBar. Compile `ProgressionBarServlet.java` and properly package files in Listing 30.13 and Listing 30.14 in root directory of the ProgressionBar Web application. Deploy the application on Tomcat 6 version. Open Internet Explorer and type http://localhost:7070/ProgressionBar/ProgressionBar.html. Figure 30.12 appears. When you click the Start button in Figure 30.12, a green progress bar is displayed:

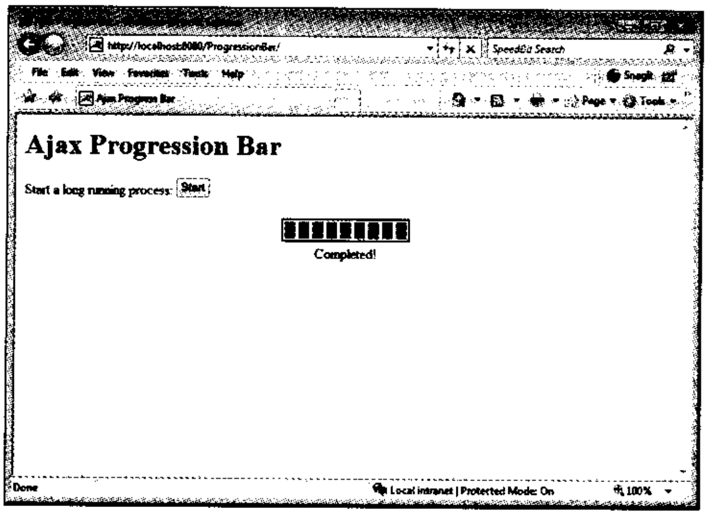

Ajax Progression Bar

Start a long running process:

64%

Figure 30.12: Showing Progression Bar

Figure 30.13 appears when the entire progress bar is displayed and the Start button becomes enabled automatically:


Ajax Progression Bar

Start a long running process: Start

Completed!

Figure 30.13: Showing Completion of Progression Bar

In this section, you have learned creating an AJAX application that contains a progress bar, which controls the other components of the application. Now, let's discuss auto complete technique of AJAX, in the following section.

Providing Auto Complete

There are many tools that display a pop up box with suggestion when you type. It speeds up data entry with greater accuracy. Google presents Google Suggest page with autocomplete. It automatically adds the most likely suggestion in text box along with the drop-down box. The drop-down box or pop-up box narrows down your search by providing related answers. The drop-down box, in case of Google, is very rich, but we have hard coded some values. Drop-down box options are displayed from these values.

The code given in Listing 30.15shows a suggest page for AutoComplete Application (you can find the autoComplete.html file in the Code/AJAX/Chapter 30/AutoComplete folder on the CD):

Listing 30.15: The autoComplete.html code of AutoComplete Application

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>Ajax Auto Complete</title>
<style type="text/css">
.mouseOut {
  background: #708090;
  color: #FFFAFA;
}
.mouseOver {
  background: #FFFAFA;
  color: #000000;
}
</style>
<script type="text/javascript">
var xmlHttp;
var completeDiv;
var inputField;
var wordTable;
var wordTableBody;
function createXMLHttpRequest() {
  if (window.ActiveXObject) {
        xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  }
  else if (window.XMLHttpRequest) {
        xmlHttp = new XMLHttpRequest();
  }
}
function initVars() {
  inputField = document.getElementById("words");
  wordTable = document.getElementById("word_table");
  completeDiv = document.getElementById("popup");
  wordTableBody = document.getElementById("word_table_body");
}
function findwords() {
  initVars();
  if (inputField.value.length > 0) {
        createXMLHttpRequest();
        var url = "AutoCompleteServlet?words=" + escape(inputField.value);
        xmlHttp.open("GET", url, true);
        xmlHttp.onreadystatechange = callback;
        xmlHttp.send(null);
  } else {
        clearwords(); }
}
function callback() {
  if (xmlHttp.readyState == 4) {
        if (xmlHttp.status == 200) {
                var word =
                xmlHttp.responseXML
                .getElementsByTagName("word")[0].firstChild.data;
                setwords(xmlHttp.responseXML.getElementsByTagName("word"));
        } else if (xmlHttp.status == 204){
                clearwords(); }
  }
}
function setWords(the_words) {
  clearWords();
  var size = the_words.length;
  setOffsets();
  var row, cell, txtNode;
  for (var i = 0; i < size; i++) {
        var nextNode = the_words[i].firstChild.data;
        row = document.createElement("tr");
        cell = document.createElement("td");
        cell.onmouseout = function() {this.className='mouseOver';};
        cell.onmouseover = function() {this.className='mouseOut';};
        cell.setAttribute("bgcolor", "#FFFAFA");
        cell.setAttribute("border", "0");
        cell.onclick = function() { populateWord(this); } ;
        txtNode = document.createTextNode(nextNode);
        cell.appendChild(txtNode);
        row.appendChild(cell);
        wordTableBody.appendChild(row);
  }
}
function setOffsets() {
  var end = inputField.offsetWidth;
  var left = calculateOffsetLeft(inputField);
  var top = calculateOffsetTop(inputField) + inputField.offsetHeight;
  completeDiv.style.border = "black 1px solid";
  completeDiv.style.left = left + "px";
  completeDiv.style.top = top + "px";
  wordTable.style.width = end + "px";
}
function calculateOffsetLeft(field) {
  return calculateOffset(field, "offsetLeft");
}
function calculateOffsetTop(field) {
  return calculateOffset(field, "offsetTop");
}
function calculateOffset(field, attr) {
  var offset = 0;
  while(field) {
        offset += field[attr];
        field = field.offsetParent;
  }
  return offset;
}
function populateWord(cell) {
  inputField.value = cell.firstChild.nodeValue;
  clearWords();
}
function clearWords() {
  var ind = wordTableBody.childNodes.length;
  for (var i = ind - 1; i >= 0 ; i--) {
        wordTableBody.removeChild(wordTableBody.childNodes[i]);
  }
  completeDiv.style.border = "none";
}
</script>
</head>
<body>
<h1>Ajax Auto Complete Example</h1>
```

```
words: <input type="text" size="20" id="words"
onkeyup="findWords();" style="height:20;"/>
<div style="position:absolute;" id="popup">
<table id="word_table" bgcolor="#FFFAFA" border="0"
cellspacing="0" cellpadding="0"/>
<tbody id="word_table_body"></tbody>
</table>
</div>
</body>
</html>
```

In the beginning of `autoComplete.html` file, the `mouseOut` and `mouseOver` are two style classes which set the background and text color when these events happen in the drop-down box area. Drop-down box is displayed in the form of a table. Variable `completeDiv` contains the whole style including top, left margins, and border settings of each word in the drop-down box. The `onkeyup` event triggers a call to `findWords()` method, which in turn calls the `initVars` method. The `initVars()` gets references to all elements in `autoComplete.html` to be populated. In findWords method, if the `inputField` has the number of letters greater than 0, then an instance of `XMLHttpRequest` object is created. For each letter to be typed, the `AutoCompleteServlet` is called and then the `escape()` function extracts the typed text by removing whitespace. The `onreadystatechange` property is initialized as call to `callback()` method. The callback method retrieves text of all word tags from XML response made by servlet, as shown in Listing 30.16. The `setWords()` method calls the `clearWords()` method, which first retrieves the number of words in row form from the drop-down box and then removes each row one by one. Now the `setOffsets()` method from setWords method is invoked, which sets the offset settings for each word; actual offsets are calculated by `calculateOffset()` method. Now from the text of all <word> tags, rows of the table are made. The <word> tag accesses the values from the ArrayList words. First of all the cells are created, then they are appended as children to rows and finally rows are appended to table body. If we click on a particular option of drop-down area, the text field word is populated with the clicked word.

The code given in Listing 30.16 shows a dynamic search for words from a word service, shown in Listing 30.17 (you can find the `AutoCompleteServlet.java` file in the Code/AJAX/Chapter 30/AutoComplete/src folder on the CD):

Listing 30.16: The `AutoCompleteServlet.java` file of AutoComplete application.

```
import java.io.*;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.servlet.*;
import javax.servlet.http.*;
public class AutoCompleteServlet extends HttpServlet {
private List words = new ArrayList();
public void init(ServletConfig config) throws ServletException {
  words.add("Aback");
  words.add("Able");
  words.add("Abstract");
  words.add("Abuse");
  words.add("Abyss");
  words.add("parse");
  words.add("palm");
  words.add("park");
  words.add("param");
}
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
  String prefix = request.getParameter("words");
  WordService service = WordService.getInstance(words);
  List matching = service.findWords(prefix);
```

```
    if (matching.size() > 0) {
          PrintWriter out = response.getWriter();
          response.setContentType("text/xml");
          response.setHeader("Cache-Control", "no-cache");
          out.println("<response>");
          Iterator iter = matching.iterator();
          while(iter.hasNext()) {
                  String word = (String) iter.next();
                  out.println("<word>" + word + "</word>");
          }
          out.println("</response>");
          matching = null;
          service = null;
          out.close();
    } else
    {
          response.setStatus(HttpServletResponse.SC_NO_CONTENT);
    }
  }
  }
```

Server-side performs dynamic lookup for words. A few words are hard coded in Servlet and the search is actually performed by `WordService.java` class. If the word being searched is not from hard coded words, the server sends the response with status code 204 indicating that there is no such content to the client.

Servlet first adds words to words list. Then, it fetches request parameter words from html page and stores it in prefix. Now the instance to words list is retrieved using `getInstance` method of `WordService.java class. From getInstance method, a call to findWords()` method is made. The `findWords()` method returns reference to list containing words that start with the entered text in the textbox of `autoComplete.html`. Each matched word is now put in one`<word>` tag and all word tags are put in `<response>` tag. Now XML response is made. If the matching list's size is equal to zero, no content response is sent to client.

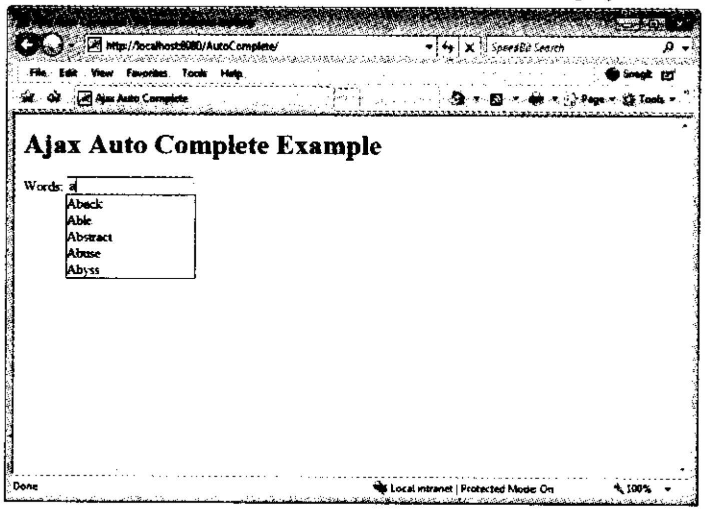
The code given in Listing 30.17 shows code for the WordService.java file (you can find this file in the Code/AJAX/Chapter 30/AutoComplete/src folder on the CD):

Listing 30.17: The WordService.java file of AutoComplete Application

```
  import java.util.ArrayList;
  import java.util.Iterator;
  import java.util.List;
  public class WordService
  {
    private List words;
    /** Creates a new instance of WordService */
    private WordService(List list_of_words)
    {
          this.words = list_of_words;
    }
    public static WordService getInstance(List list_of_words)
    {
          return new WordService(list_of_words);
    }
    public List findWords(String prefix)
    {
          String prefix_upper = prefix.toUpperCase();
          List matches = new ArrayList();
          Iterator iter = words.iterator();
          while(iter.hasNext()) {
                  String word = (String) iter.next();
                  String word_upper_case = word.toUpperCase();
                  if(word_upper_case.startsWith(prefix_upper))
```

```
                {
                        boolean result = matches.add(word);
                }
            }
            return matches;
        }
    }
```

Now make a new Web Project and name it AutoComplete. Compile `WordService.java` and `AutoCompleteServlet.java` and properly package files in Listing 30.15, 30.16, and 30.17 in the root directory AutoComplete of Web application. Deploy the application on Tomcat 6 version. Open Internet Explorer and type `http://localhost:8080/AutoComplete/`. Figure 30.14 will appear. In Figure 30.14, when you will type a character, words beginning with "a" are fetched from the server and then displayed in the drop-down box:


http://localhost:8080/AutoComplete/
File Edit View Favorites Tools Help
Ajax Auto Complete
Done
Local intranet | Protected Mode On
100%

Ajax Auto Complete Example

Words: a

Aback
Able
Abstract
Abuse
Abyss

Figure 30.14: Showing Suggestion for the Letter a

In this section, you have learned various AJAX techniques and used them to create Web applications. In addition to the XMLHttpRequest objects, you can also use IFrames to create AJAX applications. Let's discuss the differences between XMLHttpRequest and IFrames aswell how IFrames can be used to develop AJAX applications, in the following section.

XMLHttpRequest Object Vs IFrames

So far, we discussed about the XMLHttpRequest object and how it can be used for creating AJAX applications. Now we will discuss about IFrames, which is used as an alternative of XMLHttpRequest object, for AJAX applications IFrames are used for placing content from another web page onto a web page where it is used, i.e. accessing news clips, calendars, weather forecasts from another web pages or sites onto a page. IFrames are used for synchronous calls because they can load the content without reloading the entire page. The major disadvantage of IFrames is that they only support asynchronous requests. The basic structure of IFrames is as follows:

```
<iframe src="date.jsp" name="my_iframe" frameborder="0" width="500" height=
   "200" scrolling="yes" align="center" marginwidth="50", marginheight="50" ></iframe>
```

Here, the only attribute that should be specified in the <iframe> tag is src. The src attribute tells what should be displayed inside a frame. This code snippet shows that the frame sends a asynchronous request to the web page date.jsp and displays the response inside the frame. The other attributes have the following meaning:

- **Name**—Specifies the name of the frame that is used for targeting a frame.
- **Width**—Specifies the width of the frame in pixels or percentage.
- **Height**—Specifies the height of the frame in pixels or percentage.

- **Marginwidth**—Specifies the width in pixels between the frame and the left and right edges of the frame contents.
- **Marginheight**—Specifies the height in pixels between the frame and the left and right edges of the frame contents.
- **Scrolling**—Specifies whether scrolling is allowed or not. It is allowed by default. It can be set by using values yes, no, and auto.
- **Align**—Specifies the alignment of frame to the page. The possible values are top, bottom, left, right, and center.
- **Frameborder**—Specifies the size of the border in pixels outside the frame.

In this section, you have learned concepts of IFrame. Now, let's create a Web application using IFrame, in the following section.

Using IFrames

In this application, we will send an asynchronous request to the server without using XMLHttpRequest object. The application uses IFrames for sending an asynchronous request to the server. The application starts with a HTML page, index.html, and sends a request for the system's current date and time to the server. The HTML page displays a hyperlink and when you click on the hyperlink, the <iframe> tag sends an asynchronous request to the server for accessing the current date and time from the file date.jsp.

The code given in Listing 30.18 shows code for index.html page (you can find this file in the Code/AJAX/Chapter 30/IFrames folder on the CD):

Listing 30.18: The index.html File

```
<html>
<head>
<title>Using IFrames</title>
</head>
<body>
<p>Click <a target="my_iframe" href="date.jsp">here</a> and the Date and Time will
 appear in the frame below.
<p><iframe name="my_iframe" frameborder=0 width="500" height="200"></iframe></p>
</body>
</html>
```

When you open the HTML file, index.html, a screen similar to Figure 30.15, gets displayed.:

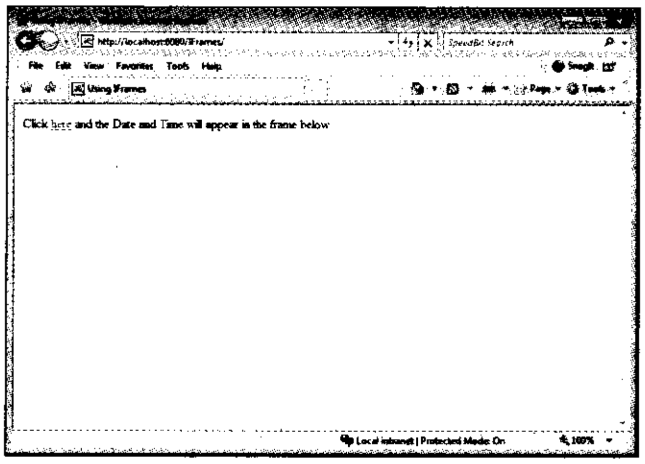

Click here and the Date and Time will appear in the frame below

Figure 30.15: Sending Asynchronous Request using IFrames

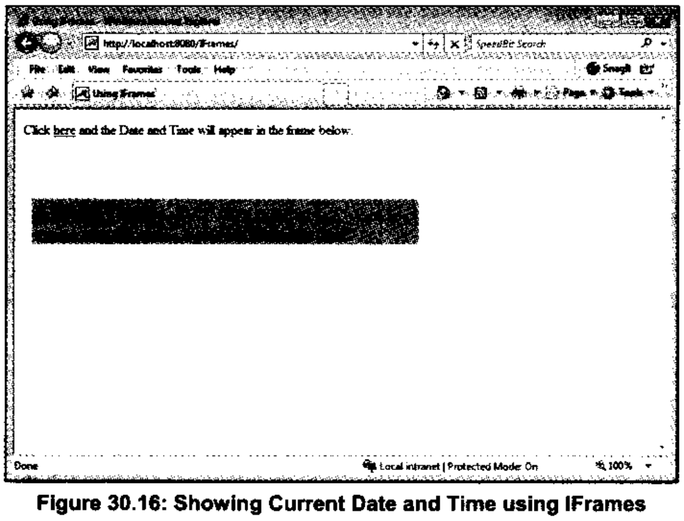
When you click on the hyperlink "here", the HTML form sends an asynchronous request to the server for accessing the current date and time from the file date.jsp.

The code given in Listing 30.19 shows code for the date.jsp file (you can find this file in the Code/AJAX/Chapter 30/IFrames folder on the CD):

Listing 30.19: The date.jsp File

```
<%@page contentType="text/html" import="java.util.*" %>
<html>
<meta http-equiv="refresh" content="5">
<body>
<p>&nbsp;</p>
<div align="left">
<table border="0" cellpadding="0" cellspacing="0" width="460" bgcolor="#ADADDE">
<tr>
<td width="100%"><font size="6" color="#000000">AJAX Date</font></td>
</tr>
<tr>
<td width="100%"><b>&nbsp;Current Date and time is:&nbsp; <font color="#FF0000">
<%= new java.util.Date()%>
</font></b></td></tr>
</table>
</div>
</body>
</html>
```

After clicking the hyperlink, the HTML page forwards the request to date.jsp page and the JSP page displays the current date and time, as shown in Figure 30.16:


http://localhost:8080/IFrames/

Click here and the Date and Time will appear in the frame below.

Figure 30.16: Showing Current Date and Time using IFrames

You can see that there is no need to reload the whole page. The entire page is reloaded without refreshing.

Now that you have gone through the entire chapter, let's take a quick summary of the topics dealt with.

Summary

Beginning with the traditional concepts, like cookies, Iframe tag, used in the websites, the chapter proceeds towards how the concept of `XMLHttpRequest` came up and how it simplified the task. The various points discussed were how the instance of `XMLHttpRequest` varies on the different browsers; the synchronous and asynchronous mode of request through the application; discussion of properties, like `onreadystatechange`, `readyState`, etc. and their implementation in an application; discussion of methods of `XMLHttpRequest` object and their implementation in an application.

In the next chapter, we will discuss about some widely used AJAX Frameworks and how these frameworks are implemented for creating AJAX applications.

Quick Revise

Q1. **Define XMLHttpRequest object.**

Ans: The concept of XMLHttpRequest object was developed by Microsoft as a part of Outlook Web Access 2000 (as the server-side API call). The Outlook Web Access 2000 was the Outlook Web-mail service which allows users to access the e-mail functionality. This was done by allowing the application to issue its own client-side HTTP requests. Then Microsoft quickly drafted it into IE5 and this is accessible through Jscript, VBScript and the other scripting languages supported by IE browsers.

Q2. **List the properties of XMLHttpRequest object.**

Ans: The following are the properties of `XMLHttpRequest` object:

- onreadystatechange property
- readyState property
- responseText property
- responseXML property
- status property
- statusText property

Q3. **Define the responseXML property of the XMLHttpRequest object.**

Ans: This property of XMLHttpRequest object returns the response as XML. This property returns an XML document object, which can be examined and parsed by using W3C DOM node tree methods and properties.

Q4. **List the XMLHttpRequest object methods.**

Ans: Following are the methods of the XMLHttpRequest object:

- `abort()`
- `open()`
- `send()`
- `setRequestHeader()`
- `getResponseHeader()`
- `getAllResponseHeaders()`

Q5. **List the states of the XMLHttpRequest object.**

Ans: The following are the states of the XMLHttpRequest object:

- 0= Uninitialized
- 1= Loading/Open
- 2= Loaded/Sent
- 3= Interactive/Receiving
- 4= Complete/Loaded

Q6. **List the parameters passed with the open() method of the XMLHttpRequest object.**

Ans: The following are the various parameters passed with the `open()` method:

- DOMString method
- DOMString URI
- Boolean async
- DOMString username
- DOMString password

Q7. **Define the send() method of XMLHttpRequest object ?**

Ans: The `send()` method sends the HTTP request to the server and receives the response. After preparing the request by using the `open()` method, it can be send to the server using send() method. The request can be send to the server when the `readyState` value is 1, otherwise an exception is raised. The request is sent to the server by using the parameters provided in the `open()` method. The `send()` method returns immediately if the async parameter has the value set as true, allowing the other client script processing to be continued.

Q8. What does the setRequestHeader() method do?

Ans: The setRequestHeader() method adds the custom HTTP headers to the request made. It sets the header to the request by adding the name and value pair to the http header to be sent. This method takes two parameters – the label and value.

Q9. How AJAX deals with browser difference?

Ans: The XMLHttpRequest object is widely used for sending AJAX request. The reason behind its popularity is that it is easy to use in a compatible way across multiple browsers. The two major browsers, Internet Explorer and Firefox, provide the same basic API. The other browsers, like Opera and Safari, also support the same basic API, but only in their more recent versions.

Q10. List basic AJAX techniques used for Web application development.

Ans: The basic AJAX techniques used for Web application development are:

- Performing validation
- Reading response headers
- Loading list boxes dynamically
- Auto refreshing page
- Using dynamic progress bar
- Providing auto complete