69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Lecture Sheet on "PHP+MySQL"
Day-18: (JSON-PHP)

Working with JavaScript **Time: 2** hours

This lecture provides the learner with information about implementing JavaScript for handling forms with effective form validation

Objectives	Topics
❖ Learn the basics of JavaScript ❖ Know the syntax, rules and conventions ❖ Handling different fields and forms	❖ What is JavaScript ❖ Syntax, Variables, Keywords ❖ Reserved words ❖ Data conversion ❖ Form Basics ❖ Handling fileds with JavaScript

Overview

With the still-rising popularity of web services, XML has practically become the de facto standard for data transmission. However, XML is not without its detractors. For example, some consider it to be overly verbose for data transmission purposes, necessitating many more bytes of information to be sent across the Internet to accomplish what could have been done in a much smaller form. To take this into consideration, new forms of XML compression and even entire new XML formats, such as Binary XML, have been developed. All these solutions work on extending or adding on to XML, making backwards compatibility an issue. **Douglas Crockford**, a long-time software engineer, proposed a new data format built on JavaScript called **JavaScript Object Notation (JSON)**. Up until this point, you've used either plain text or XML to transmit data back and forth to the server. This chapter introduces the use of JSON in Ajax communication as an alternative to these more traditional data formats.

What Is JSON?

JSON is a very lightweight data format based on a subset of the JavaScript syntax, **namely array and object literals**. Because it uses JavaScript syntax, JSON definitions can be included within JavaScript files and accessed without the extra parsing that comes along with XML- based languages. But before you can use JSON, it's important to understand the specific JavaScript syntax for array and object literals.

Array Literals

For those unfamiliar with JavaScript literal notation, array literals are specified by using square brackets ([and]) to enclose a comma-delimited list of JavaScript values (meaning a string, number, Boolean, or null value), such as:

```
var aNames = ["Benjamin", "Michael", "Scott"];
```

You can then access the values in the array by using the array name and bracket

notation:

```
alert(aNames[0]);          //outputs "Benjamin"
alert(aNames[1]);          //outputs "Michael"
alert(aNames[2]);          //outputs "Scott"
```

Note that the first position in the array is 0, and the value in that position is "Benjamin". Because arrays in JavaScript are not typed, they can be used to store any number of different data types: `var aValues = ["string", 24, true, null];`

This array contains a string, followed by a number, followed by a Boolean, followed by a null value. This is completely legal and perfectly fine JavaScript. If you were to define an array without using literal notation, you would have to use the Array constructor, such as:

```
var aValues = new Array("string", 24, true, null);
```

It's important to note that either way is acceptable when writing JavaScript, but only array literals are valid in JSON. Important

Object Literals

Object literals are used to store information in name-value pairs, ultimately creating an object. An object literal is defined by two curly braces ({ and }). Inside of these can be placed any number of name-value pairs, defined with a string, a colon, and the value. Each name-value pair must be followed with a comma, except for the last one (making this more like defining an associative array in Perl). For example:

```
var oCar = {
        "color" : "red",
        "doors" : 4,
        "paidFor" : true
};
```

This code creates an object with three properties named color, doors, and paidFor, each

containing different values. You can access these properties by using the object name and dot notation, such as:

```
alert(oCar.color);          //outputs "red"
alert(oCar.doors);          //outputs "4"
alert(oCar.paidFor);        //outputs "true"
```

You can also use bracket notation and pass in the name of the property as a string value (like the way it was defined using object literal notation):

```
alert(oCar["color"]);       //outputs "red"
alert(oCar["doors"]);       //outputs "4"
alert(oCar["paidFor"]);     //outputs "true"
```

The same object could be created using the JavaScript Object constructor, like this:

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
var oCar = new Object();
oCar.color = "red";
oCar.doors = 4;
oCar.paidFor = true;
```

As you can see, the object literal notation requires much less code than using the Object constructor.Once again, although either approach is valid in JavaScript, only object Important literal notation is valid in JSON.

Mixing Literals

It's possible to mix object and array literals, creating an array of objects or an object containing an array. Suppose you wanted to create an array of car objects similar to the one created in the last section. You could do so as follows:

```
var aCars = [
        {
            "color" : "red",
            "doors" : 2,
            "paidFor" : true
        },
        {
            "color" : "blue",
            "doors" : 4,
            "paidFor" : true
        },
        {
            "color" : "white",
            "doors" : 2,
            "paidFor" : false
        }
];
```

This code defines an array, aCars, which has three objects in it. The three objects each have properties named color, doors, and paidFor. (Each object represents a car, of course.) You can access any of the information in the array by using a combination of bracket and dot notation. For example, to get the number of doors on the second car in the array, the following line will do the trick:

```
alert(aCars[1].doors);          //outputs "4"
```

In this example, you are first getting the value in the second position of the array (position 1) and then getting the property named doors. You can also define the array to be inside of object literals, such as:

```
var oCarInfo = {
        "availableColors" : [ "red", "white", "blue" ],
        "availableDoors" : [ 2, 4 ]
};
```

This code defines an object called oCarInfo that has two properties, availableColors and availableDoors. Both of these properties are arrays, containing strings and numbers, respectively. To access a value here, you just reverse the order of the bracket and dot notation. So, to get to the second available color, you can do this:

```
alert(oCarInfo.availableColors[1]);
```

In this example, you are first returning the property named availableColors, and then getting the value in the second position (position 1). But what does all this have to do with JSON?

JSON Syntax

JSON syntax is really nothing more than the mixture of object and array literals to store data. The only difference from the examples in the last section is that JSON doesn't have variables. Remember, JSON represents only data; it has no concept of variables, assignments, or equality. Therefore, the JSON code for the last example is simply:

```
{
        "availableColors" : [ "red", "white", "blue" ],
        "availableDoors" : [ 2, 4 ]
}
```

Note that the variable oCarInfo has been removed, as has the semicolon following the closing

curly brace. If this were transmitted via HTTP to a browser, it would be fairly quick because of the small number of characters. Suppose this data was retrieved by using XMLHTTP (or some other form of client-server communication) and stored in a variable named sJSON. You now have a string of information, not an object and certainly not an object with two arrays. To transform it into an object, you can simply use the JavaScript eval() function, like so:

```
var oCarInfo = eval("(" + sJSON + ")");
```

This example surrounds the JSON text with parentheses and then passes that string into the eval() function, which acts like a JavaScript interpreter. The result of this operation is a JavaScript object identical to the oCarInfo object defined in the last section. You can access information in this object in the exact same way:

```
alert(oCarInfo.availableColors[0]);          //outputs "red"
alert(oCarInfo.availableDoors[1]);           //ouputs "4"
```

It's very important to include the extra parentheses around any JSON Important string

before passing it into eval(). Remember, curly braces also represent statements in JavaScript (such as used with the if statement). The only way the interpreter knows that the curly braces represent an object and not a statement is to look for an equals sign before it or to look for parentheses around it (which indicates that the code is an expression to be evaluated instead of a statement to be run). You can see the obvious benefits of using JSON as a data format for JavaScript communication: it takes the evaluation of the data out of your hands and, therefore, grants you faster access to the information contained within.

JSON Encoding/Decoding

As part of his resource on JSON, Crockford has created a utility that can be used to decode and encode between JSON and JavaScript objects. The source for this tool can be downloaded at www.crockford.com/JSON/json.js.

You may be asking yourself, "Didn't I just learn to decode JSON using the eval() function?" The answer is yes, but there is an inherent flaw in using eval(): it will evaluate any JavaScript code passed into it, not just JSON. This could be a huge security risk when dealing with enterprise web applications. To deal with this issue, you can use the JSON.parse() method (defined in the aforementioned file), which will parse and convert only JSON code into JavaScript. For example:

```
var oObject = JSON.parse(sJSON);
```

Also provided with the utility is a way to convert JavaScript objects into a JSON string for transmission. (There is no such built-in way to do this in JavaScript.) All you need to do is pass an object into the JSON.stringify() method. Consider the following example:

```
var oCar = new Object();
oCar.doors = 4;
oCar.color = "blue";
oCar.year = 1995;
oCar.drivers = new Array("Penny", "Dan", "Kris");
document.write(JSON.stringify(oCar));
```

This code outputs the following JSON string:

```
{"doors":4,"color":"blue","year":1995,"drivers":["Penny","Dan","Kris"] }
```

With this, you're now ready to transmit the information to whatever destination makes sense for you.

JSON versus XML

As mentioned previously, one of the advantages of JSON over XML is that it's more compact. XML is considered by some to be overly verbose for its purpose. But what does this mean exactly? Consider the following XML data:

```
<classinfo>
```

```
<students>
    <student>
        <name>Michael Smith</name>
        <average>99.5</average>
        <age>17</age>
        <graduating>true</graduating>
    </student>
    <student>
        <name>Steve Johnson</name>
        <average>34.87</average>
        <age>17</age>
        <graduating>false</graduating>
    </student>
    <student>
        <name>Rebecca Young</name>
        <average>89.6</average>
        <age>18</age>
        <graduating>true</graduating>
    </student>
</students>
</classinfo>
```

This example contains information about three students in a class. There is some XML information that isn't entirely necessary: the <classinfo> and <students/> elements. These elements help to define the overall structure and meaning of the information, but the actual information you're interested in is the students and their information. Plus, for each piece of information about the students, the name of the information is repeated twice, although the actual data is repeated only once (for example, "name" appears both in <name> and </name>. Consider the same information formatted as JSON:

```
{ "classinfo" :
    {
        "students" : [
            {
                "name" : "Michael Smith",
                "average" : 99.5,
                "age" : 17,
                "graduating" : true
```

```
                },
                {
                        "name" : "Steve Johnson",
                        "average" : 34.87,
                        "age" : 17,
                        "graduating" : false
                },
                {
                        "name" : "Rebecca Young",
                        "average" : 89.6,
                        "age" : 18,
                        "graduating" : true
                }
        ]
    }
}
```

As you can see, a lot of the superfluous information isn't present. Since closing tags aren't necessary to match opening tags, it greatly reduces the number of bytes needed to transmit the same information. Not including spaces, the JSON data is 224 bytes, whereas the comparable XML data is 365 bytes, saving more than 100 bytes. (This is why Crockford, JSON's creator, calls it the "fat free alternative to XML.") The disadvantage to JSON-formatted data as compared to XML is that it's less readable to the layperson. Because XML is verbose, it's fairly easy to understand what data is being represented. JSON, with its shorthand notation, can be difficult to decipher using the naked eye. Of course, an argument can be made that data exchange formats should never be viewed with the naked eye. If you're using tools to create and parse the data being passed back and forth, then there is really no reason to have the data be human readable. But this begs the question: Are there any JSON tools available? The answer is yes.

Server-Side JSON Tools

As you know by now, Ajax has to do with the interaction between the client and the server, so JSON would be of no use for this purpose unless there were server-side tools to aid in the encoding and decoding. As luck would have it, there are quite a few JSON utilities for server- side languages. Although it is beyond the scope of this book to discuss every one of these tools, it is useful to take a look at one and then develop a solution using it.

JSON-PHP

JSON-PHP is a PHP utility to ease the encoding and decoding of JSON information.

This utility, written by Michal Migurski, is available for free at http://mike.teczno.com/json.html. All you need to begin using JSON in PHP is to include the JSON.php file in your page and make use of the JSON object. Creating new instance of the JSON object is quite simple:

```
<?php
	require_once("JSON.php");
	$oJSON = new JSON();
?>
```

The first line includes the JSON.php file that contains the JSON object definition. The second line simply instantiates the object and stores it in the variable $oJSON. Now you're ready to start encoding and decoding JSON in your PHP page. To encode a PHP object into a JSON string, use the encode() method, which accepts a single argument: an object to encode, which can be an array or a full-fledged object. It doesn't matter how the object or array was created, whether using a class definition or not; all objects can be encoded using this method. Consider the following class definition:

```
<?php
	class Person {
		var $age;
		var $hairColor;
		var $name;
		var $siblingNames;
		function Person($name, $age, $hairColor) {
			$this->name = $name;
			$this->age = $age;
			$this->hairColor = $hairColor;
			$this->siblingNames = array();
		}
	}
?>
```

This PHP code defines a class called Person that stores some personal information. You would use this class as follows:

```
<?php
	$oPerson = new Person("Mike", 26, "brown");
	$oPerson->siblingNames[0] = "Matt";
	$oPerson->siblingNames[1] = "Tammy";
?>
```

To encode the $oPerson object, you simply pass it into the encode() method, like

this:

```
<?php     $sJSONText = $oJSON->encode($oPerson); ?>
```

This creates a JSON string of:

{"age":26,"hairColor":"brown","name":"Mike","siblingNames":["Matt","Ta mmy"]}

The $oPerson object is now ready to be transferred to JavaScript or any other language that can support JSON-encoded information. But what if you already have a JSON string? That's where the decode() method is used. Suppose you have the JSON string displayed previously and want to create a PHP object from it. Just pass the string into the decode() method:

```
<?php    $oPerson = $oJSON->decode($sJSONText); ?>
```

Now the $oPerson variable can be used just like the one in the previous example, as if it were created using the Person class:

```
<?php
        print("<h3>Person Information</h3>");
        print("<p>Name: ".$oPerson->name."<br />");
        print("Age: ".$oPerson->age."<br />");
        print("Hair Color: ".$oPerson->hairColor."<br />");
        print("Sibling Names:</p><ul>");
        for ($i=0; $i < count($oPerson->siblingNames); $i++) {
                print("<li>".$oPerson->siblingNames[$i]."</li>");
        }
        print("</ul>");
?>
```

This code prints out the information contained in the $oPerson object, proving that the object has been constructed appropriately. JSON-PHP will be used in several projects throughout this book because it is quite simply the easiest way to deal with JSON in a server-side language.

Creating an Autosuggest Text Box

The best way to learn about any new programming concept is to put it into a practical example. Google Suggest (located at www.google.com/webhp?complete=1) is a very simple Ajax application that many programmers have spent time dissecting, analyzing, and re-creating. If you haven't yet taken a look at the live application, please do so now; it will greatly aid in your understanding of the following example. Functionality such as this, suggesting to the user values to type in, has been around in desktop applications for some time now. Google Suggest brought the idea to the Web and generated a lot of excitement while doing it. As mentioned earlier in the book, Google Suggest really was one of the early Ajax applications that got developers excited about the concept. It seems fitting to attempt to emulate the behavior of

Google Suggest to help others understand Ajax. The example built in this section aids in the selection of states or provinces in a personal information form. For sites that deal with international customers, it is often vital to include the state or province along with the country. However, it's not optimal to load every state and province in the entire world into a drop-down box for the user to select from. It's much easier to let the user start typing, and then retrieve only those results that would make the most sense. Autosuggest functionality is perfect for this use case.

Functionality Overview

Before building anything, it's always helpful to understand exactly what you're building. Anyone can say they are going to emulate the functionality of Google Suggest, but what does that mean? The example you will build in this section has the following functionality:

Typeahead: As the user is typing, the rest of the text box fills in with the best suggestion at the time. As the user continues to type, the text box automatically adjusts its suggestion. The suggested text always appears selected (highlighted). This should work no matter how fast the user types. Suggestions list: Also as the user is typing, a drop-down list of other suggestions is displayed. These suggestions are generated automatically while the user types so that there is no discernible delay. Keyboard controls: When the suggestions are displayed, the user is able to scroll up and down the list by using the up and down arrows on the keyboard and select a suggestion. Pressing Enter places the value into the text box and hides the suggestion list. The Esc key can also be used to hide the suggestions. Hide suggestions: The drop-down suggestion list is smart enough to hide itself whenever the text box is not used or when the browser window is hidden. You probably didn't realize that Google Suggest was doing so much. This is the key with Ajax: you don't think about what's going on because it works in an intuitive way.

The HTML

The first step in any client-side component is to build the HTML that will be used. For the autosuggest text box, this includes the text box itself as well as the drop-down list of suggestions. You're probably familiar with the HTML text box:

```
<input type=" text" name=" txtAutosuggest" value="" />
```

In most cases, this line would be enough to use a text box. The problem is that some browsers (notably Internet Explorer on Windows and Mozilla Firefox on all operating systems) provide autocomplete functionality that drops down a list of suggestions based on values you've entered before. Since this would compete directly with the suggestions you'll be providing, this has to be turned off. To do so, set the autocomplete attribute to off:

```
<input type=" text" name=" txtAutosuggest" value="" autocomplete=" off" />
```

Now you can be assured that there will be no interference from the autocomplete

browser behavior. The only other user interface component to design is the drop-down list of suggestions. The suggestion drop-down list is nothing more than an absolutely positioned element that is positioned below the text box so as to give the illusion of being a drop-down list. Inside of this element are several other

elements, one for each suggestion. By changing the style of these elements, it's possible to achieve the look of highlighting a given suggestion. The HTML to create the list displayed in Figure is as follows:

```
<div class=" suggestions">
        <div class=" current">Maine</div>
        <div>Maryland</div>
        <div>Massachusetts</div>
        <div>Michigan</div>
        <div>Minnesota</div>
        <div>Mississippi</div>
        <div>Missouri</div>
        <div>Montana</div>
</div>
```

This HTML won't be coded directly into the main HTML file; instead, it will be created dynamically by JavaScript code. However, you need to know the general format of the HTML in order to create it appropriately. Of course, some CSS is needed to make the drop-down list function properly. The outermost has a class of suggestions, which is defined as:

```
div.suggestions {
        -moz-box-sizing: border-box;
        box-sizing: border-box;
        background-color: white;
        border: 1px solid black;
        position: absolute;
}
```

The first two lines of this CSS class are for browsers that support two forms of box sizing: content box and border box (for more information, read www.quirksmode.org/css/box.html). In quirks mode, Internet Explorer defaults to border box; in standards mode, Internet Explorer defaults to content box. Most other DOM-compliant browsers (Mozilla, Opera, and Safari) default to content box, meaning that there is a difference in how the element will be rendered among browsers. To provide for this, the first two lines of the CSS class set rendering to border box. The first line, -moz-box-sizing, is Mozilla-specific and used for older

Mozilla browsers; the second line is for browsers that support the official CSS3 box-sizing property. Assuming that you use quirks mode in your page, this class will work just fine. (If you use standards mode, simply remove these first two lines.) The remaining styles simply add a border and specify that the element be absolutely positioned. Next, a little bit of formatting is needed for the drop-down list items:

```
div.suggestions div {
    cursor: default;
    padding: 0px 3px;
}
```

The first line specifies the default cursor (the arrow) to be displayed when the mouse is over an item in the drop-down list. Without this, the cursor would display as the caret, which is the normal cursor for text boxes and web pages in general. The user needs to believe that the drop- down item is not a part of the regular page flow, but an attachment to the text box, and changing the cursor helps. The second line simply applies some padding to the item (which you can modify as you wish). Last, some CSS is needed to format the currently selected item in the drop-down list. When an item is selected, the background will be changed to blue and the text color will be changed to white. This provides a basic highlight that is typically used in drop-down menus:

```
div.suggestions div.current {
    background-color: #3366cc;
    color: white;
}
```

All these styles are to be contained in an external CSS file named autosuggest.css.

The Database Table

In order to easily query the states and provinces that match a particular text snippet, it is necessary to use a database table. There are a few open source database products that can be paired nicely with PHP, but the most common one is MySQL (available at www.mysql.org). Many web hosting companies that offer PHP hosting also offer one or more MySQL databases as well, so for this example, MySQL will be the database used. Although this example is intended to be used with MySQL, it should be Important able to run on other databases with little or no modification. The database table can be very simple for this example, although you may need more information to make it practical for your needs. To get this to work, you really need only a single column to store the state and province names. However, it's always best to define a primary key, so this table will include a second column containing an auto-incremented ID number for each state or province. The following SQL statement creates a table named StatesAndProvinces with two columns, Id and

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Name:

```
CREATE TABLE StatesAndProvinces (
    Id INT NOT NULL AUTO_INCREMENT,
    Name VARCHAR(255) NOT NULL,
    PRIMARY KEY (Id)
) COMMENT = 'States and Provinces';
```

Of course, the time-consuming part is to fill in state and province names from various countries around the world. The code download for this example, available at www.wrox.com, includes a SQL file that will populate the table with all U.S. states as well as one that will insert all Canadian provinces and territories. Setting up this information in a database table enables you to quickly get a list of suggestions for text the user has typed in. If the user has typed the letter M, for example, you can run the following query to get the first five suggestions:

```
SELECT *
FROM StatesAndProvinces
WHERE Name LIKE 'M%'
ORDER BY Name ASC
LIMIT 0, 5
```

This statement returns a maximum of five suggestions, in alphabetical order, for all names starting with M. Later, this will be used in the PHP code that returns the suggestions. **Architecture**

In Lecture-1, you saw the basic architecture of an Ajax solution involving the user interface and Ajax engine on the client. The autosuggest architecture follows this general format, where the user interface is the autosuggest control and the Ajax engine is a suggestion provider.

In this architecture, the autosuggest control has no idea where the suggestions are coming from; they could be coming from the client or the server. All the autosuggest control knows is how to call the suggestion provider to get suggestions for the text contained within the text box. The suggestion provider handles all the server communication and notifies the autosuggest control when the suggestions are available. To accomplish this, both the autosuggest control and the suggestion provider need to implement specific interfaces so that each knows what method to call on the other.

The Classes

Two classes are necessary to represent the two client-side components of the autosuggest functionality, appropriately called AutoSuggestControl and SuggestionProvider. The AutoSuggestControl is assigned a SuggestionProvider when it is created so that all requests go through it. The SuggestionProvider has only one

method, requestSuggestions(), which is called by the AutoSuggestControl whenever suggestions are needed. This method takes two arguments: the AutoSuggestControl itself and a Boolean value indicating whether the control should type ahead when the suggestions are returned. When the suggestions have been retrieved, the SuggestionProvider calls the autosuggest() method of the AutoSuggestControl, passing in the array of suggestions as well as the typeahead flag that was passed into it. This allows for a delay between the request for suggestions and the response, making it possible to use asynchronous requests. This approach sounds more complicated than it is; Figure below represents the interaction between these two objects in a clearer manner.

With the architecture designed, it's time to start coding. The AutoSuggest Control
The AutoSuggestControl class is the wrapper for all autosuggest functionality. To work properly, the control needs to know which text box to work on and the suggestion provider to use. This makes for a relatively simple constructor:

```
function AutoSuggestControl(oTextbox, oProvider) {
        this.provider = oProvider;

        this.textbox = oTextbox;

}
```

It's upon this simple base that the complex functionality of an autosuggest text box will be built. Since the AutoSuggestControl class is quite complicated, it's much simpler to break up its explanation into specific types of functionality. The following sections build on each other, and the complete code can be downloaded from www.wrox.com.

Implementing Typeahead

To implement the first feature, typeahead, it's helpful to understand exactly how it works. Typeahead text boxes look at what the user has typed and then makes a suggestion, highlighting only the part that was added automatically. For example, if you were to type Ma into a text box, the suggestion may be Maine, but only ine would be highlighted. Doing this allows the user to continue typing without interruption because any new characters simply replace the highlighted section. Originally, the only type of highlighting possible using JavaScript was to highlight all the text in the text box using the select() method, as follows:

```
var oTextbox = document.getElementById("txtState");
oTextbox.select();
```

This code gets a reference to a text box with the ID of txtState and then selects all the text contained within it. Although this functionality is fine for many everyday uses, it's not very helpful for implementing typeahead. Thankfully, both Internet Explorer and Firefox have ways of selecting parts of the text instead of the entire item (for other browsers, this feature is not available). But as usual, the two biggest combatants in the browser world do things in two completely different ways. The

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Internet Explorer solution is to use a text range. Not to be confused with DOM ranges, an Internet Explorer text range is an invisible selection of text on the page, beginning on a single character and ending on a single character. When a text range is filled out, you can highlight just the text contained within it, which is perfect for typeahead. To create a text range for a specific text box, you use the createTextRange() method that Internet Explorer provides on every text box. Once you have a text range, its methods enable you to select certain parts of the text. Although there are many text range methods, the only ones of interest for this example are moveStart() and moveEnd(), both of which accept two arguments: a unit and a number. The unit can be character, word, sentence, or textedit, whereas the number indicates the number of units to move from the start or end of the text (this should be a positive number for moveStart() and a negative for moveEnd()). When the endpoints of the text range are set, you can call its select() method to highlight just those characters. For example, to select just the first three characters in a text box, you could do this:

```
var oRange = oTextbox.createTextRange();
oRange.moveStart("character", 0);

oRange.moveEnd("character", 3 - oTextbox.value.length);

oRange.select();

oTextbox.focus();
```

Note that to get the appropriate value for moveEnd(), it's necessary to subtract the length of the text in the text box from the number of characters to select (3). The last step is to set the focus to the text box so that the selection is visible. (Text can be selected only when the text box has focus.) The process is a bit involved in Internet Explorer, but pretty easy to script. Firefox, on the other hand, is very straightforward. Text boxes in Firefox have a non-standard method called setSelectionRange(), which accepts two arguments: the index of the character to start with and the index of character after the last character to select. So, to select the first three characters in a text box using Mozilla, you need only two lines of code:

```
oTextbox.setSelectionRange(0,3);

oTextbox.focus();
```

The first method you'll need in the AutoSuggestControl class is a method to select a range of characters in a browser-specific way. This method, called selectRange(), handles all the dirty work for you:

```
AutoSuggestControl.prototype.selectRange = function (iStart, iEnd) {
    if (this.textbox.createTextRange) {
        var oRange = this.textbox.createTextRange();
        oRange.moveStart("character", iStart);
```

```
                oRange.moveEnd("character", iEnd- this.textbox.value.length);
                oRange.select();
        } else if (this.textbox.setSelectionRange) {
                this.textbox.setSelectionRange(iStart, iEnd);
        }
        this.textbox.focus();
};
```

This method uses feature detection, the process of detecting certain browser features, to determine how to select the characters. It tests for the existence of the createTextRange() method to determine whether the Internet Explorer text ranges should be used, and tests for the setSelectionRange() method to determine whether the Firefox method should be used. The arguments are the first character to select and the number of characters to select. These values are then passed to the browser-specific methods of text selection.

The typeAhead() Method

Now that you can select specific parts of the text box, it's time to implement the typeahead functionality. To do this, a typeAhead() method is defined that accepts a single argument: the suggestion to display in the textbox. The suggestion being passed in is assumed to be appropriate (and assumed to have at least one character). This method then does three things:

1. It gets the length of the text already in the text box.
2. It places the suggestion into the text box.
3. It selects only the portion of the text that the user didn't type using the information from

step 1.

Additionally, since typeahead can be supported only in Internet Explorer and Firefox, you should check to make sure one of those browsers is being used. If the browser doesn't support text selection, then none of the steps should be executed so as not to interrupt the user's typing. Once again, testing for the createTextRange() and setSelectionRange() methods of the text box is the way to go:

```
AutoSuggestControl.prototype.typeAhead = function (sSuggestion) {
        if (this.textbox.createTextRange ||
this.textbox.setSelectionRange) {
                var iLen = this.textbox.value.length;
                this.textbox.value = sSuggestion;
                this.selectRange(iLen, sSuggestion.length);
        }
};
```

With this method complete, you now need another method to call it and pass in the

suggestion. This is where the autosuggest() method comes in.

The autosuggest() Method

Perhaps the most important method in the control is autosuggest(). This single method is responsible for receiving an array of suggestions for the text box and then deciding what to do with them. Eventually, this method will be used to implement the full autosuggest functionality (including drop-down suggestions), but for now, it's used to implement typeahead only. Because autosuggest() will be passed an array of suggestions, you have your pick as to which one to use for the typeahead value. It's recommended to always use the first value in the array to keep it simple. The problem is that there may not always be suggestions for a value, in which case an empty array will be passed. You shouldn't call typeAhead() if there are no suggestions, so it's important to check the length of the array first:

```
AutoSuggestControl.prototype.autosuggest = function (aSuggestions) {
        if (aSuggestions.length > 0) { this.typeAhead(aSuggestions[0]); }

};
```

But where do the suggestions come from? It's actually the job of the suggestion provider to call this method and pass in the suggestions.

Handling Key Events

Of course the autosuggest functionality has to be tied into the text box using events. There are three events that deal with keys: keydown, keypress, and keyup. The keydown event fires whenever the user presses a key on the keyboard but before any changes occur to the text box. This obviously won't help with autosuggest because you need to know the full text of the text box; using this event would mean being one keystroke behind. For the same reason, the keypress event can't be used. It is similar to key-down, but fires only when a character key is pressed. The keyup event, however, fires after changes have been made to the text box, which is exactly when autosuggest should begin working. Setting up an event handler for the text box involves two steps: defining a function and assigning it as an event handler. The function is actually a method of the autosuggest control, called handleKeyUp(). This method expects the event object to be passed in as an argument (how to accomplish this is discussed later) so that it can tell whether the key being pressed should enact the autosuggest functionality. Since keyup fires for all keys, not just character keys, you'll receive events when someone uses a cursor key, the Tab key, and any other key on the keyboard. To avoid interfering with how a text box works, suggestions should be made only when a character key is pressed. This is where the event object's keyCode property enters the picture. The keyCode property is supported by most modern browsers (including Internet Explorer on Windows and Macintosh, Firefox, Opera, and Safari) and returns a numeric code representing the key that was pressed. Using this property, it's possible to set up behaviors for specific keys. Since the autosuggest functionality should happen only when character keys are pressed, you

need to check this property for an appropriate value before proceeding. Believe it or not, the easiest way to do this is actually to detect the keys that you want to ignore. This approach is more efficient because there are more character keys than non-character keys. The following table displays the key codes for all keys that should be ignored:

<ferdous>

You may notice a pattern among the key codes. It looks like all keys with a code less than or equal to 46 should be ignored and all keys with codes between 112 and 123 should be ignored.

This is generally true, but there is an exception. The space bar has a key code of 32, so you actually need to check to see if the code is less than 32, between 33 and 46, or between 112 and 123. If it's not in any one of these groups, then you know it's a character key.

Here's what the handleKeyUp() method looks like:

```
AutoSuggestControl.prototype.handleKeyUp = function (oEvent) {
        var iKeyCode = oEvent.keyCode;
           if (iKeyCode < 32 || (iKeyCode >= 33 && iKeyCode <= 46)
                 || (iKeyCode >= 112 && iKeyCode <= 123)) {
                //ignore
        } else { this.provider.requestSuggestions(this); }
};
```

When a user presses a character key, the autosuggest functionality begins by calling the suggestion provider's requestSuggestions() method and passing a pointer to the autosuggest control as an argument. Remember, it's the suggestion provider that will call the autosuggest() method defined earlier. The requestSuggestions() method begins the process of retrieving suggestions for usage.

With this method defined, it must be assigned as the event handler for the text box. It's best to create a separate method to handle initializations for the control such as this (there will be more in the future). The init() method serves this purpose:

```
AutoSuggestControl.prototype.init = function () {
        var oThis = this;
        this.textbox.onkeyup = function (oEvent) {
                if (!oEvent) {    oEvent = window.event; }
                oThis.handleKeyUp(oEvent);
        };
};
```

The init() method starts by creating a pointer to the this object so that it may be used

later. An anonymous function is defined for the text box's onkeyup event handler. Inside of this function, the handleKeyUp() method is called using the oThis pointer. (Using this here would refer to the text box instead of the autosuggest control.) Since this method requires the event object to be passed in, it's necessary to check for both DOM and Internet Explorer event objects. The DOM event object is passed in as an argument to the event handler, whereas the Internet Explorer event object is a property of window. Instead of doing a browser detect, you can check to see if the oEvent object is passed into the event handler. If not, then assign window.event into the oEvent variable.

The oEvent variable can then be passed directly into the handleKeyUp() event handler. The init() method should be called from within the AutoSuggestControl constructor:

```
function AutoSuggestControl(oTextbox, oProvider) {
        this.provider = oProvider;
        this.textbox = oTextbox;
        this.init();
}
```

That's all it takes to implement the typeahead functionality of the autosuggest control. At this point, you are displaying a single suggestion to the user as they type. The goal is, of course, to provide multiple suggestions using a drop-down list.

Showing Multiple Suggestions

Earlier in the chapter you took a look at the HTML and CSS used for the drop-down list of suggestions. Now the task is to create the HTML programmatically and apply the CSS to create the actual functionality; this is a multistep process. First, a property is needed to store the element because various methods of the AutoSuggestControl need access to it.

This property is called layer and is initially set to null:

```
function AutoSuggestControl(oTextbox, oProvider) {
        this.layer = null;
        this.provider = oProvider;
        this.textbox = oTextbox;
        this.init();
}
```

The drop-down list will be created after you define a few simple methods to help control its behavior. The simplest method is hideSuggestions(), which hides the drop-down list after it has been shown:

```
AutoSuggestControl.prototype.hideSuggestions = function () {
        this.layer.style.visibility = "hidden";
```

```
};
```

Next, a method is needed for highlighting the current suggestion in the drop-down list. The highlightSuggestion() method accepts a single argument, which is the element containing the current suggestion. The purpose of this method is to set the element's class attribute to current on the current suggestion and clear the class attribute on all others in the list. Doing so provides a highlighting effect on the drop-down list similar to the regular form controls. The algorithm is quite simple: iterate through the child nodes of the layer.

If the child node is equal to the node that was passed in, set the class to current; otherwise, clear the class attribute by setting it to an empty string:

```
AutoSuggestControl.prototype.highlightSuggestion = function (oSuggestionNode) {
        for (var i=0; i < this.layer.childNodes.length; i++) {
                var oNode = this.layer.childNodes[i];
                if (oNode == oSuggestionNode) {
                        oNode.className = "current"
                } else if (oNode.className == "current") {
                        oNode.className = "";
                }
        }
};
```

With these two methods defined, it's time to create the drop-down list . The createDropDown() method creates the outermost element and defines the event handlers for the drop-down list. To create the element, use the createElement() method and then assign the various styling properties:

```
AutoSuggestControl.prototype.createDropDown = function () {
        this.layer = document.createElement("div");
        this.layer.className = "suggestions";
        this.layer.style.visibility = "hidden";
        this.layer.style.width = this.textbox.offsetWidth;
        document.body.appendChild(this.layer);
        //more code to come
};
```

This code first creates the element and assigns it to the layer property. From there, the className (equivalent to the class attribute) is set to suggestions, as is needed for the CSS to work properly. The next line hides the layer, since it should be invisible initially. Then, the width of the layer is set equal to the width of the text box by using the text box's offsetWidth property (this is optional depending on your individual needs). The very last line adds the layer to the document. With the layer

created, it's time to assign the event handlers to control it.

At this point, the only concern is making sure that the drop-down list is functional if the user uses the mouse. That is, when the drop-down list is visible, moving the mouse over a suggestion should highlight it. Likewise, when a suggestion is clicked on, it should be placed in the text box and the drop-down list should be hidden. To make this happen, you need to assign three event handlers: onmouseover, onmousedown, and onmouseup.

The onmouseover event handler is used simply to highlight the current suggestion; onmousedown is used to select the given suggestion (place the suggestion in the text box and hide the drop-down list); and onmouseup is used to set the focus back to the text box after a selection has been made. Because all these events are fired by the drop-down list itself, it's best just to use a single function for all of them, as follows:

```
AutoSuggestControl.prototype.createDropDown = function () {
        this.layer = document.createElement("div");
        this.layer.className = "suggestions";
        this.layer.style.visibility = "hidden";
        this.layer.style.width = this.textbox.offsetWidth;
        document.body.appendChild(this.layer);
        var oThis = this;
        this.layer.onmousedown = this.layer.onmouseup =
        this.layer.onmouseover = function (oEvent) {
                oEvent = oEvent || window.event;
                oTarget = oEvent.target || oEvent.srcElement;
                if (oEvent.type == "mousedown") {
                        oThis.textbox.value = oTarget.firstChild.nodeValue;
                        oThis.hideSuggestions();
            } else if (oEvent.type == "mouseover") { oThis.highlightSuggestion(oTarget);
                } else { oThis.textbox.focus(); }
        };
};
```

The first part of this section is the assignment of oThis equal to the this object. This is necessary so that a reference to the AutoSuggestControl object is accessible from within the event handler. Next, a compound assignment occurs, assigning the same function as an event handler for onmousedown, onmouseup, and onmouseover. Inside of the function, the first two lines are used to account for the different event models (DOM and IE), using a logical OR (||) to assign the values for oEvent and oTarget. (The target will always be a element containing a suggestion.)

If the event being handled is mousedown, then set the value of the text box equal to the text inside of the event target. The text inside of the element is contained in a text node, which is the first child node. The actual text string is contained in the text node's nodeValue property. After the suggestion is placed into the text box, the drop-down list is hidden.

When the event being handled is mouseover, the event target is passed into the highlightSuggestion() method to provide the hover effect; when the event is mouseup, the focus is set back to the text box (this fires immediately after mousedown).

Positioning the Drop-Down List

To get the full effect of the drop-down list, it's imperative that it appears directly below the text box. If the text box were absolutely positioned, this wouldn't be much of an issue. In actual practice, text boxes are rarely absolutely positioned and more often are placed inline, which presents a problem in aligning the drop-down list. To calculate the position where the drop- down list should appear, you can use the text box's offsetLeft, offsetTop, and offsetParent properties.

The offsetLeft and offsetTop properties tell you how many pixels away from the left and top of the offsetParent an element is placed. The offsetParent is usually, but not always, the parent node of the element, so to get the left position of the text box, you need to add up the offsetLeft properties of the text box and all of its ancestor elements (stopping at <body/>),

as seen below:

```
AutoSuggestControl.prototype.getLeft = function () {
        var oNode = this.textbox;

        var iLeft = 0;

        while(oNode.tagName != "BODY") {

                iLeft += oNode.offsetLeft;

                oNode = oNode.offsetParent;

        }

        return iLeft;

};
```

The getLeft() method begins by pointing oNode at the text box and defining iLeft with an initial value of 0. The while loop will continue to add oNode.offsetLeft to iLeft as it traverses up the DOM structure to the <body/> element.

The same algorithm can be used to get the top of the text box:

```
AutoSuggestControl.prototype.getTop = function () {
        var oNode = this.textbox;

        var iTop = 0;
```

```
        while(oNode.tagName != "BODY") {
                iTop += oNode.offsetTop;
                oNode = oNode.offsetParent;
        }
        return iTop;
};
```

These two methods will be used to place the drop-down list in the correct location.

Adding and Displaying Suggestions

The next step in the process is to create a method that adds the suggestions into the drop-down list and then displays it. The showSuggestions() method accepts an array of suggestions as an argument and then builds up the necessary DOM elements to display them. From there, the method positions the drop-down list underneath the text box and displays it to the user:

```
AutoSuggestControl.prototype.showSuggestions = function (aSuggestions)
{
        var oDiv = null;
        this.layer.innerHTML = "";
        for (var i=0; i < aSuggestions.length; i++) {
                oDiv = document.createElement("div");
                oDiv.appendChild(document.createTextNode(aSuggestions[i]));
                this.layer.appendChild(oDiv);
        }
        this.layer.style.left = this.getLeft() + "px";
        this.layer.style.top = (this.getTop()+this.textbox.offsetHeight) +
"px";
        this.layer.style.visibility = "visible";
};
```

The first line simply defines the variable oDiv for later use. The second line clears the contents of the drop-down list by setting the innerHTML property to an empty string. Then, the for loop creates a element and a text node for each suggestion before adding it to the drop- down list layer.

The next section of code starts by setting the left position of the layer using the getLeft()

method. To set the top position, you need to add the value from getTop() to the height of the text box (retrieved by using the offsetHeight property). Without doing this, the drop-down list would appear directly over the text box. (Remember, getTop() retrieves the top of the text box, not the top of the drop-down list layer.)

Last, the layer's visibility property is set to visible to show it.

Updating the Functionality

In order to show the drop-down list of suggestions, you'll need to make several changes to the functionality defined previously.

The first update is the addition of a second argument to the autosuggest() method, which indicates whether the typeahead functionality should be used (the reason why will be explained shortly). Naturally, the typeAhead() method should be called only if this argument is true. If there's at least one suggestion, typeahead should be used and the drop-down list of suggestions should be displayed by calling showSuggestions(); if there's no suggestions, the drop-down list should be hidden by calling hideSuggestions():

```
AutoSuggestControl.prototype.autosuggest = function (aSuggestions,
bTypeAhead) {
        if (aSuggestions.length > 0) {
                if (bTypeAhead) { this.typeAhead(aSuggestions[0]); }
                this.showSuggestions(aSuggestions);
        } else {    this.hideSuggestions();        }
};
```

It's also necessary to update the handleKeyUp() method for a couple of different reasons. The first reason is to add the bTypeAhead argument to the requestSuggestions() call. When called from here, this argument will always be true:

```
AutoSuggestControl.prototype.handleKeyUp = function (oEvent) {
        var iKeyCode = oEvent.keyCode;
        if (iKeyCode < 32 || (iKeyCode >= 33 && iKeyCode <= 46)
                || (iKeyCode >= 112 && iKeyCode <= 123)) {
                //ignore
        } else {    this.provider.requestSuggestions(this, true);          }
};
```

Remember, the requestSuggestions() method is defined on the suggestion provider, which is described later in this chapter.

This functionality now works exactly as it did previously, but there are a couple of other keys that require special attention: Backspace and Delete. When either of these keys is pressed, you don't want to activate the typeahead functionality because it will disrupt the process of removing characters from the text box. However, there's no reason not to show the drop-down list of suggestions. For the Backspace (key code of 8) and Delete (key code of 46) keys, you can call requestSuggestions(), but this time, pass in false to indicate that typeahead should not occur:

```
AutoSuggestControl.prototype.handleKeyUp = function (oEvent) {
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
        var iKeyCode = oEvent.keyCode;
        if (iKeyCode == 8 || iKeyCode == 46) {
                this.provider.requestSuggestions(this, false);
        } else if (iKeyCode < 32 || (iKeyCode >= 33 && iKeyCode <= 46)
                                || (iKeyCode >= 112 && iKeyCode <= 123)) {
                //ignore
        } else {                this.provider.requestSuggestions(this, true);          }
};
```

Now when the user is removing characters, suggestions will still be displayed and the user can click one of them to select the value for the text box. This is acceptable, but to really be usable the autosuggest control needs to respond to keyboard controls.

Adding Keyboard Support

The desired keyboard functionality revolves around four keys: the up arrow, the down arrow, Esc, and Enter (or Return). When the drop-down suggestion list is displayed, the user should be able to press the down arrow to highlight the first suggestion, then press it again to move to the second, and so on. The up arrow should then be used to move back up the list of suggestions.

As each suggestion is highlighted, the value must be placed in the text box. If the user presses Esc, the suggestions should be hidden and the suggestion should be removed from the text box. When the Enter key is pressed, the suggestions should also be hidden, but the last suggestion should remain highlighted in the text box.

In order for the user to use the up and down arrow keys, you'll need to keep track of the currently selected item in the suggestions list. To do this, you must add two properties to the AutoSuggestControl definition, as follows:

```
function AutoSuggestControl(oTextbox, oProvider) {
        this.cur = -1;
        this.layer = null;
        this.provider = oProvider;
        this.textbox = oTextbox;
        this.userText = oTextbox.value;
        this.init();
}
```

The cur property stores the index of the current suggestion in the suggestions array. By

default, this value is set to -1 because there are no suggestions initially. When the arrow keys are pressed, cur will change to point to the current suggestion. The second added property, userText, holds the current value of the text box and changes

to reflect what the user actually typed.

As cur changes, the highlighted suggestion changes as well. To encapsulate this functionality, a method called goToSuggestion() is used. This method accepts only one argument, a number whose sign indicates which direction to move in. For instance, any number greater than 0 moves the selection to the next suggestion; any number less than or equal to 0 moves the selection to the previous suggestion. Here's the code:

```
AutoSuggestControl.prototype.goToSuggestion = function (iDiff) {
        var cSuggestionNodes = this.layer.childNodes;
        if (cSuggestionNodes.length > 0) {
                var oNode = null;
                if (iDiff > 0) {
                   if (this.cur < cSuggestionNodes.length-1) { oNode = cSuggestionNodes[++this.cur]; }
                } else {
                        if (this.cur > 0) {    oNode = cSuggestionNodes[--this.cur];                    }
                }
                if (oNode) {
                        this.highlightSuggestion(oNode);
                        this.textbox.value = oNode.firstChild.nodeValue;
                }
        }
};
```

This method begins by obtaining the collection of child nodes in the drop-down layer. Since only elements containing suggestions are child nodes of the layer, the number of child nodes accurately matches the number of suggestions. This number can be used to determine if there are any suggestions (in which case it will be greater than zero). If there are no suggestions, the method need not do anything.

When there are suggestions, a variable named oNode is created to store a reference to the

suggestion node to highlight, and the method checks to see which direction to go in. If iDiff is greater than 0, it tries to go to the next suggestion. In doing so, the method first checks to ensure that cur isn't greater than the number of suggestions minus 1 (because the index of the last element in a collection with n elements is n-1). Assuming there is a next suggestion, cur is prefix incremented (meaning it assumes its new value before the line it's on executes) to retrieve the node for the next suggestion.

If iDiff is less than or equal to zero, then that means the previous suggestion needs to be highlighted. In that case, you must first check to ensure cur is greater than 0 (if cur

isn't at least 1, then there isn't a previous suggestion to go to). Passing that test, cur is then prefix decremented to get a reference to the correct suggestion node.

The last step in the method is to ensure that oNode isn't null. If it's not, then the node is passed to highlightSuggestion() and the suggestion text is placed into the text box; if it is null, then no action is taken.

Another part of keeping track of the selected suggestion is to be sure cur is reset at the correct point; otherwise, you can get some very odd behavior. The correct place to reset cur to –1 is in the autosuggest() method, just before the drop-down list is displayed:

```
AutoSuggestControl.prototype.autosuggest = function (aSuggestions,
bTypeAhead){
        this.cur = -1;
        if (aSuggestions.length > 0) {
                if (bTypeAhead) {               this.typeAhead(aSuggestions[0]);        }
                this.showSuggestions(aSuggestions);
        } else {        this.hideSuggestions();         }
};
```

Along the same lines, it's important to set userText to the correct value. This should be done in the handleKeyUp() method:

```
AutoSuggestControl.prototype.handleKeyUp = function (oEvent) {
        var iKeyCode = oEvent.keyCode;
        this.userText = this.textbox.value;
        if (iKeyCode == 8 || iKeyCode == 46) {
                this.provider.requestSuggestions(this, false);
        } else if (iKeyCode < 32 || (iKeyCode >= 33 && iKeyCode <= 46)
                        || (iKeyCode >= 112 && iKeyCode <= 123)) {
                //ignore
        } else {
                this.provider.requestSuggestions(this, true);
        }
};
```

This small addition saves what the user typed before asking for suggestions. This will be very useful when dealing with the Esc key. With these two methods updates, all that's left is to make sure that goToSuggestion() gets called at the right time.

To handle the up arrow, down arrow, Esc, and Enter keys, a handleKeyDown() method is necessary. Similar to handleKeyUp(), this method also requires the event object to be passed in. And once again, you'll need to rely on the key code to tell

which key was pressed. The key codes for the up arrow, down arrow, Esc, and Enter keys are 38, 40, 27, and 13, respectively. The handleKeyDown() method is defined as follows:

```
AutoSuggestControl.prototype.handleKeyDown = function (oEvent) {
        switch(oEvent.keyCode) {
                case 38: //up arrow
                        this.goToSuggestion(-1);
                        break;
                case 40: //down arrow
                        this.goToSuggestion(1);
                        break;
                case 27: //esc
                        this.textbox.value = this.userText;
                        this.selectRange(this.userText.length, 0);
                        /* falls through */
                case 13: //enter
                        this.hideSuggestions();
                        oEvent.returnValue = false;
                        if (oEvent.preventDefault) {
                                oEvent.preventDefault();
                        }
                        break;
        }
};
```

When the up arrow is pressed (key code 38), the goToSuggestion() method is called with an argument of –1, indicating that the previous selection should be selected. Likewise, when the down arrow is pressed (key code 40), goToSuggestion() is called with 1 as an argument to highlight the next suggestion. If Esc is pressed (key code 27), there are a couple of things to do. First, you need to set the text box value back to the original text that the user typed. Second, you need to set the selection of the text box to be after what the user typed so that he or she can continue typing. This is done by setting the selection range to the length of the text with a selection length of zero. Then, this case falls through to the Enter key's case (key code 13), which hides the suggestions list. This way, the code contains only one call to hideSuggestions() instead of two. Remember, when the user presses the up or down arrows, the suggestion is automatically placed into the text box. This means that when the Enter key is pressed, you need only hide the drop-down list of suggestions. For both Esc and Enter, you also must block the default behavior for the key press. This is

important to prevent unintended behavior, such as the Enter key submitting the form when the user really just wanted to select the current suggestion. The default behavior is blocked first by setting event.returnValue equal to false (for IE) and then calling preventDefault() (if it's available, for Firefox). Updating init() Now that all this new functionality has been added, it must be initialized. Previously, the init() method was used to set up the onkeyup event handler; now it must be extended to also set up the onkeydown and onblur event handlers, as well as to create the drop-down suggestion list. The onkeydown event handler is set up in a similar manner as onkeyup:

```
AutoSuggestControl.prototype.init = function () {
        var oThis = this;
        this.textbox.onkeyup = function (oEvent) {
                if (!oEvent) { oEvent = window.event; }
                oThis.handleKeyUp(oEvent);
        };
        this.textbox.onkeydown = function (oEvent) {
                if (!oEvent) { oEvent = window.event; }
                oThis.handleKeyDown(oEvent);
        };
        //more code to come
};
```

As you can see, the same algorithm is used with the onkeydown event handler: first, determine the location of the event object, and then pass it into the handleKeyDown() method. Up to this point, the only way the drop-down list is hidden is when the user presses the Enter key. But what if the user clicks elsewhere on the screen or uses the Tab key to switch to a new form field? To prepare for this event, you must set up an onblur event handler, which hides the suggestions whenever the text box loses focus:

```
AutoSuggestControl.prototype.init = function () {
        var oThis = this;
        this.textbox.onkeyup = function (oEvent) {
                if (!oEvent) { oEvent = window.event; }
                oThis.handleKeyUp(oEvent);
        };
        this.textbox.onkeydown = function (oEvent) {
                if (!oEvent) { oEvent = window.event; }
                oThis.handleKeyDown(oEvent);
        };
        this.textbox.onblur = function () { oThis.hideSuggestions(); };
```

```
        this.createDropDown();
};
```

You'll also notice that the createDropDown() method is called to create the initial drop-down list structure. This completes the keyboard support for the autosuggest control, but there is one more thing to take into account.

Fast-Type Support

Because the handleKeyUp() method requests suggestions whenever a key is pressed, you may be wondering if it can keep up when someone is typing quickly. The answer is no. You may be surprised to know that it is possible to type too fast for the event handling to keep up. In this case, you get suggestions that are too late (including letters you never typed) and a very choppy user experience (with long pauses as you type). So, how can you make sure that fast typists aren't left out of this functionality?

Quite simply, you should wait a short amount of time before requesting suggestions from the server. This can be done using the setTimeout() method, which delays the calling of a

function for a set time interval. The new functionality works like this: a timeout ID is saved in the AutoSuggestControl object. If another key is pressed before the timeout has been activated, the existing timeout is cleared and a new one is put in its place. So basically, when a user presses a key, the control waits a certain amount of time before requesting suggestions. If another key is pressed before the request is made, the control cancels the original request (by clearing the timeout) and asks for a new request to be made after the same amount of time. In this way, you can be sure that the request for suggestions goes out only while the user has paused typing.

To implement this functionality, the first thing you need is a property to hold the timeout ID. You can add the timeoutId property directly to the AutoSuggestControl class, as follows:

```
function AutoSuggestControl(oTextbox, oProvider) {
        this.cur = -1;
        this.layer = null;
        this.provider = oProvider;
        this.textbox = oTextbox;
       this.timeoutId = null;
       this.userText = oTextbox.value;
       this.init();
}
```

Next, update the handleKeyUp() method to make use of this new property:

```
AutoSuggestControl.prototype.handleKeyUp = function (oEvent
/*:Event*/) {
```

```
        var iKeyCode = oEvent.keyCode;
        var oThis = this;
        this.userText = this.textbox.value;
        clearTimeout(this.timeoutId);
        if (iKeyCode == 8 || iKeyCode == 46) {
                this.timeoutId = setTimeout(function () {
                        oThis.provider.requestSuggestions(oThis, false);
                }, 250);
        } else if (iKeyCode < 32 || (iKeyCode >= 33 && iKeyCode < 46)
                                || (iKeyCode >= 112 && iKeyCode <= 123)) {
                //ignore
        } else {
                this.timeoutId = setTimeout(function () {
                        oThis.provider.requestSuggestions(oThis, true);
                }, 250);
        }
};
```

The first new line in this method stores a reference to the this object, which is important when using the setTimeout() method. The second new line of code clears any timeout that may have already been started; this cancels and suggestion request that may have been initiated.

The other two sections of new code change the call to the requestSuggestions() to occur after 250 milliseconds (which is plenty of time for this purpose). Each call is wrapped in an anonymous function that is passed into setTimeout(). The result of setTimeout(), the timeout ID is stored in the new property for later usage. All in all, this ensures that no requests will be made unless the user has stopped typing for at least 250 milliseconds.

This completes the code for the AutoSuggestControl class. All of the functionality has been implemented, and all that's left is to create a suggestion provider to call.

The Suggestion Provider

The SuggestionProvider class is relatively simple compared to the AutoSuggestControl since it has only one purpose: to request suggestions from the server and forward them to the control. To do so, SuggestionProvider needs an instance of XmlHttp. Instead of using a new object for each request, the same object will be used over and over, to avoid the overhead of creating and destroying objects in rapid succession. This single instance is created using the zXML library's zXmlHttp factory and stored in a property called http:

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

```
function SuggestionProvider() {
        this.http = zXmlHttp.createRequest();
}
```

The lone method of the suggestion provider is requestSuggestions(), which you may remember from the architecture discussion. This method accepts two arguments: the AutoSuggestControl to work on and a flag indicating whether typeahead should be used.

The complete code is as follows:

```
SuggestionProvider.prototype.requestSuggestions = function (oAutoSuggestControl, bTypeAhead) {
        var oHttp = this.http;
        //cancel any active requests
        if (oHttp.readyState != 0) {          oHttp.abort();          }
        //define the data
        var oData = {
                requesting: "StatesAndProvinces",
                text: oAutoSuggestControl.userText,
                limit: 5
        };
        //open connection to server
        oHttp.open("post", "suggestions.php", true);
        oHttp.onreadystatechange = function () {
                if (oHttp.readyState == 4) {
                        //evaluate the returned text JavaScript (an array)
                        var aSuggestions = JSON.parse(oHttp.responseText);
                        //provide suggestions to the control
                        oAutoSuggestControl.autosuggest(aSuggestions, bTypeAhead);
                }
        };
         //send the request
         oHttp.send(JSON.stringify(oData));
};
```

The first line inside the method sets oHttp equal to the stored XmlHttp object. This is done simply for convenience and keeping the code clean. Next, you check to make sure that there isn't already a request waiting for a response. If the XmlHttp object is ready to be used cleanly, its readyState will be 0; otherwise, you must cancel the existing request (by calling abort()) before making another request.

Because the data being sent to the server is to be JSON-encoded, you first need to

create an object (oData) to hold the information. There are three pieces of information being sent: the table to get the data out of, the current value in the text box, and the maximum number of suggestions to retrieve (5). The maximum number of suggestions is important because it prevents long database queries from being executed repeatedly.

Next, a request is opened to suggestions.php, the server-side component of the control.

This request is asynchronous (last argument of open() is set to true), so it's necessary to provide an onreadystate-change event handler. The event handler first checks to ensure that the readyState is 4, and then parses the returned text as a JSON array of values. This array, along with the original typeahead flag, is then passed back to the AutoSuggestControl via the autosuggest() method.

The last step in this method is, of course, to send the request. Note that since the request is doing a POST, the data has to be passed into the send() method. The oData object is first encoded into JSON before being sent.

With that, the SuggestionProvider class is complete. The only thing left to do is to write the suggestions.php file that uses the data that is sent.

The Server-Side Component

In many ways, the server-side component for the autosuggest control is the most straightforward: it's just a single thread being executed from top to bottom, with no functions or methods to be concerned about. Note that because this is a PHP page, all the code discussed in this section must be contained within a PHP code block (<?php . . . ?>).

The first part of the page is to set the content type to text/plain, indicating that this is a plain text file and shouldn't be handled as anything else. You can optionally specify a character set, but make sure that it is Unicode-compatible, such as UTF-8, since all Unicode characters are valid in JavaScript. Here's the line that accomplishes assigning the content type:

```
header("Content-Type: text/plain; charset=UTF-8");
```

Next, include the JSON-PHP library and create a new instance of the JSON object:

```
require_once("JSON.php");

$oJSON = new JSON();
```

Normally when data is sent to a PHP page, you can use $_GET, $_POST, or $_REQUEST to retrieve it. In this case, however, the data isn't being sent in traditional name/value pairs;

instead, it's being sent as a JSON string, and there is no built-in support for this specific type of data. Instead, you need to get the body of the request and decode it manually. The body of any request is available in PHP through

$HTTP_RAW_POST_DATA, which contains the original, encoded content that was sent. Because the JSON string wasn't URL encoded, however, you can just pass this directly into the decode() method to reconstitute the oData object:

```
$oData = $oJSON->decode($HTTP_RAW_POST_DATA);
```

You'll also need an array to store the suggestions in:

```
$aSuggestions = array();
```

If there are no suggestions, no values will be added to the array and an empty array ([]) will be returned to the client. Before tapping the database for suggestions, make sure that there is actually text in the text box. Suggestions are requested when the user hits Delete or Backspace, so there's a possibility that the text box could be empty. You should check for this first by seeing if the length of the text is greater than 0; if so, you can continue on to query the database.

The query string itself is built up from the data submitting from the client. The name of the table, the LIKE statement, and the number of results to return are all incorporated into the SQL query.

The following code creates a connection to the database, executes the query, and then adds the results of the query to the $aSuggestions array:

```
if (strlen($oData->text) > 0) {
        //create the SQL query string
        $sQuery = "Select Name from ".$oData->requesting." where Name like '".
                        $oData->text."%' order by Name ASC limit 0,".$oData- >limit;
        //make the database connection
        $oLink = mysql_connect($sDBServer,$sDBUsername,$sDBPassword);
        @mysql_select_db($sDBName) or die("Unable to open database");
        if($oResult = mysql_query($sQuery)) {
                while ($aValues = mysql_fetch_array($oResult,MYSQL_ASSOC)) {
                        array_push($aSuggestions, $aValues['Name']);
                }
        }
        mysql_free_result($oResult);
        mysql_close($oLink);
}
```

This code should be fairly familiar to you as it is the same basic algorithm used throughout the book to access a MySQL database using PHP. (You must fill in the appropriate values for $sDBServer, $sDBUsername, and $sDBPassword to reflect your database settings.) The only unique part is that the results are being stored in an array, which facilitates the conversion into a JSON string to be sent back to the client.

The actual encoding is the very last step of the page. In one step, you can encode the array and output it to the page:

echo($oJSON->encode($aSuggestions));

Now it's up to the client to parse the JSON code correctly.

The Client-Side Component

So far, you've built the HTML, CSS, JavaScript, and PHP to be used by the autosuggest control.

The only thing left to do is to assemble it all into a page that you can use. The most important thing to remember is the inclusion of all necessary JavaScript files. In this case, you need to include json.js, zxml.js, and autosuggest.js. Also important is the inclusion of the style sheet file, autosuggest.css.

It's also necessary to instantiate the AutoSuggestControl after the page has completely loaded, using the onload event handler. The complete code for the example page is:

```
<html><head>
        <title>Autosuggest Example</title>
        <script type=" text/javascript" src=" json.js"></script>
        <script type=" text/javascript" src=" zxml.js"></script>
        <script type=" text/javascript" src=" autosuggest.js"></script>
        <link rel=" stylesheet" type=" text/css" href=" autosuggest.css" />
        <script type=" text/javascript">
            window.onload = function () {
var oTextbox = new
AutoSuggestControl(document.getElementById("txtState"), new SuggestionProvider());
            }
        </script>
    </head>
    <body>
        <form method=" post" action=" your_action.php">
            <table border="0">
                <tr>
                    <td>Name:</td>
                    <td><input type=" text" name=" txtName" id=" txtName" /></td>
                </tr>
                <tr>
                    <td>Address 1:</td>
```

```
<td><input type=" text" name=" txtAddress1" id=" txtAddress1" /></td>
                    </tr>
                    <tr>
                        <td>Address 2:</td>
<td><input type=" text" name=" txtAddress2" id=" txtAddress2" /></td>
                    </tr>
                    <tr>
                        <td>City:</td>
                       <td><input type=" text" name=" txtCity" id=" txtCity" /></td>
                   </tr>
                   <tr>
                        <td>State/Province:</td>
<td><input type="text" name="txtState" id=" txtState"
autocomplete="off"/></td>
                   </tr>
                   <tr>
                        <td>Zip Code:</td>
                        <td><input type=" text" name=" txtZip" id=" txtZip" /></td>
                   </tr>
                   <tr>
                        <td>Country:</td>
 <td><input type=" text" name=" txtCountry" id="txtCountry" /></td>
                   </tr>
              </table>
              <input type=" submit" value=" Save Information" />
         </form>
    </body></html>
```

Note that once the necessary files are included, you need to place only one line of JavaScript in the window.onload event handler to set up the functionality:

```
var oTextbox = new
AutoSuggestControl(document.getElementById("txtState"),
                                                  new SuggestionProvider());
```

This line creates a new AutoSuggestControl object, passing a reference to the text box with the id of txtState and a new SuggestionProvider() class. It's important that this line be executed in the onload event handler because document.getElementById() isn't 100 percent accurate until the entire page has been loaded.

The example itself is done in a way in which this control may be used: filling in personal information. This could be a page where customers can update their information or it could be a shipping form. Whichever way you choose to use this functionality, it is sure to improve the usability of your form. An autosuggest control, although not as flashy as some Ajax solutions, really is a good example of how Ajax can be used in a non-interfering way.

Summary

In this chapter, you learned all about JavaScript Object Notation (JSON) as an alternative data transmission format to XML. You learned that JSON has several advantages over XML for data transmission needs, including a smaller amount of code to represent the same data and a logical object-and-array structure that most programming languages can understand and use.

You also learned that while JavaScript can understand and interpret JSON natively, there are several server-side libraries that provide the same functionality. You learned about a JavaScript utility for parsing and encoding JSON data, as well as the JSON-PHP library that can be used to do the same for PHP.

The chapter went on to describe how to make an Ajax-assisted autosuggest control that enables you to display suggestions based on what the user has typed. This control works

similar to the way that Google Suggest does and takes into account user interaction with the mouse and keyboard as well as providing for fast typists. This control helped to illustrate the power of simple Ajax solutions.

The next chapter will expand on what you've learned here to create reusable Ajax widgets for your web site. These widgets can use a variety of data transmission formats, including JSON.