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

Lecture Sheet on "PHP+MySQL"
Day-16: (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

JavaScript is a scripting language most often used for client-side web development. It was the originating dialect of the ECMAScript standard. It is a dynamic, weakly typed, prototype-based language with first-class functions. JavaScript was influenced by many languages and was designed to look like Java, but be easier for non-programmers to work with.

Although best known for its use in websites (as client-side JavaScript), JavaScript is also used to enable scripting access to objects embedded in other applications.

JavaScript, despite the name, is essentially unrelated to the Java programming language, although both have the common C syntax, and JavaScript copies many Java names and naming conventions. The language was originally named "LiveScript" but was renamed in a co-marketing deal between Netscape and Sun, in exchange for Netscape bundling Sun's Java runtime with their then-dominant browser. The key design principles within JavaScript are inherited from the Self programming language.

"JavaScript" is a trademark of Sun Microsystems. It was used under license for technology invented and implemented by Netscape Communications and current entities such as the Mozilla Foundation.

JavaScript Implementations

Although ECMAScript is an important standard, it is not the only part of JavaScript, and certainly not the only part that has been standardized. Indeed, a complete JavaScript implementation is made up of

- ❑ The Core (ECMAScript)
- ❑ The Document Object Model (DOM)
- ❑ The Browser Object Model (BOM)

Syntax

Developers familiar with languages such as Java, C, and Perl will find ECMAScript syntax easy to pick up because it borrows syntax from each. Java and ECMAScript have several key syntax features in common, as well as some that are completely different.

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

The basic concepts of ECMAScript are the following:

- **Everything is case-sensitive**. Just as with Java, variables, function names, operators, and everything else is case-sensitive, meaning that a variable named test is different from one named Test.
- **Variables are loosely typed**. Unlike Java and C, variables in ECMAScript are not given a specific type. Instead, each variable is defined using the var operator and can be initialized with any value. This enables you to change the type of data a variable contains at any point in time (although you should avoid doing so whenever possible). Some examples:

```
var color = "red";
var num = 25;
var visible = true;
```

- **End-of-line semicolons are optional**. Java, C, and Perl require that every line end with a semicolon (;) to be syntactically correct; ECMAScript allows the developer to decide whether or not to end a line with a semicolon. If the semicolon is not provided, ECMAScript considers the end of the line as the end of the statement (similar to Visual Basic and VBScript), provided that this doesn't break the semantics of the code. Proper coding practice is to always include the semicolons because some browsers won't run properly without them, but according to the letter of the ECMAScript standard, both of the following lines are proper syntax:

```
var test1 = "red"
var test2 = "blue";
```

- **Comments are the same as in Java, C, and Perl**. ECMAScript borrowed its comments from these languages. There are two types of comments: single-line and multiline. The single-line comments begin with two forward-slashes (//), whereas multiline comments begin with a forward-slash and asterisk (/*) and end with an asterisk followed by a forward-slash (*/).

```
//this is a single-line comment
/* this is a multi-
line comment */
```

- **Braces indicate code blocks**. Another concept borrowed from Java is the code block. Code blocks are used to indicate a series of statements that should be executed in sequence and are indicated by enclosing the statements between an opening brace ({) and a closing brace (}). For example:

```
if (test1 == "red") {        test1 = "blue";        alert(test1);    }
```

Variables

As I mentioned, variables in ECMAScript are defined by using the var operator (short for variable), followed by the variable name, such as: `var test = "hi";`

In this example, the variable test is declared and given an initialization value of "hi" (a string).

Because ECMAScript is loosely typed, the interpreter automatically creates a string

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

value for test without any explicit type declaration. You can also define two or more variables using the same var statement: var test = • ghi • h, test2 = • ghola • h;

Also unlike Java, variables can hold different types of values at different times; this is the advantage of loosely typed variables. A variable can be initialized with a string value, for instance, and later on be set to a number value, like this:

```
var test = "hi";
alert(test);          //outputs "hi"
//do something else here
test = 55;
alert(test);          //outputs "55"
```

This code outputs both the string and the number values without incident (or error). As mentioned previously, it is best coding practice for a variable to always contain a value of the same type throughout its use.

In terms of variables names, a name must follow two simple rules:

- ❑ The first character must be a letter, an underscore (_), or a dollar sign ($).
- ❑ All remaining characters may be underscores, dollar signs, or any alphanumeric characters.

All the following variable names are legal:

```
var   test;
var   $test;
var   $1;
var   _$te$t2;
```

- ❑ **Camel Notation** • \ the first letter is lowercase and each appended word begins with an uppercase letter. For example: var myTestValue = 0, mySecondTestValue = • ghi • h;

- ❑ **Pascal Notation** • \ the first letter is uppercase and each appended word begins with an uppercase letter. For example: var MyTestValue = 0, MySecondTestValue = • ghi • h;

- ❑ **Hungarian Type Notation** • \ prepends a lowercase letter (or sequence of lowercase letters) to the beginning of a Pascal Notation variable name to indicate the type of the variable. For example, i means integer and s means string in the following line:

```
var iMyTestValue = 0, sMySecondTestValue = "hi";
```

The following table list prefixes for defining ECMAScript variables with Hungarian Type Notation.

These prefixes are used throughout the book to make sample code easier to read:

Another interesting aspect of ECMAScript (and a major difference from most programming languages)is that variables don't have to be declared before being used. For example:

```
var sTest = "hello ";
sTest2 = sTest + "world";
alert(sTest2);          //outputs "hello world"
```

Keywords

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

ECMA-262 describes a set of keywords that ECMAScript supports. These keywords indicate beginnings and/or endings of ECMAScript statements. By rule, keywords are reserved and cannot be used as variable or function names. Here is the complete list of ECMAScript keywords:

break	else	new	var	case	finally	return	void	catch
for	switch	while	continue	function	this	with	default	if
throw	delete	in	try	do	instanceof	typeof		

If you use a keyword as a variable or function name, you will probably be greeted with an error message like this: "Identifier expected."

Reserved Words

ECMAScript also defines a number of reserved words. The reserved words are, in a sense, words that are reserved for future use as keywords. Because of this, reserved words cannot be used as variable or function names. The complete list of reserved words in ECMA-262 Edition 3 is as follows:

abstract	enum	int	short	boolean	export	interface	static	byte
extends	long	super	char	final	native	synchronized	class	
float	package	throws	const	goto	private	transient	debugger	
implements	protected	volatile	double	import	public			

The typeof operator

The typeof operator takes one parameter: the variable or value to check. For example:

```
var sTemp = "test string";
alert(typeof sTemp);          //outputs "string"
alert(typeof 95);          //outputs "number"
```

Calling typeof on a variable or value returns one of the following values:

- "undefined" if the variable is of the Undefined type.
- "boolean" if the variable is of the Boolean type.
- "number" if the variable is of the Number type.
- "string" if the variable is of the String type.
- "object" if the variable is of a reference type or of the Null type.

Converting to a string

The interesting thing about ECMAScript primitive values for Booleans, numbers, and strings is that they are pseudo-objects, meaning that they actually have properties and methods. For example, to get the length of a string, you can do the following:

```
var sColor = "blue";
alert(sColor.length);                //outputs "4"
```

Even though the value "blue" is a primitive string, it still has a length property holding the size of the string. To that end, the three main primitive values, **Booleans**, **numbers**, and **strings**, all have a toString() method to convert their value to a string.

You may be asking, "Isn't it ridiculously redundant to have a toString() method for a string?" Yes, it is. But ECMAScript defines all objects, whether they are pseudo-objects representing primitive values or full-fledged objects, to have a toString() method. Because the string type falls in the category of **pseudo-object**, it also must have a toString() method.

The Boolean toString() method simply outputs the string "true" or "false", depending on the value of the

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

variable:

```
var bFound = false;
alert(bFound.toString());          //outputs "false"
```

The Number toString() method is unique in that it has two modes: default and radix mode. In default mode, the toString() method simply outputs the numeric value in an appropriate string (whether that is integer, floating point, or e-notation), like this:

```
var iNum1 = 10;
var fNum2 = 10.0;
alert(iNum1.toString());          //outputs "10"
alert(fNum2.toString());          //outputs "10"
```

Converting to a number

ECMAScript provides two methods for converting non-number primitives into numbers: **parseInt()** and **parseFloat()**. As you may have guessed, the former converts a value into an integer whereas the latter converts a value into a floating-point number. These methods only work properly when called on strings; all other types return NaN.

Both parseInt() and parseFloat() look at a string carefully before deciding what its numeric value should be. The parseInt() method starts with the character in position 0 and determines if this is a valid number; if it isn't, the method returns NaN and doesn't continue. If, however, the number is valid, the method goes on to the character in position 1 and does the same test. This process continues until a character isn't a valid number, at which point parseInt() takes the string (up to that point) and converts it into a number.

For example, if you want to convert the string "1234blue" to an integer, parseInt() would return a value of 1234 because it stops processing one it reaches the character b.

Any number literal contained in a string is also converted correctly, so the string "0xA" is properly converted into the number 10. However, the string "22.5" will be converted to 22, because the decimal point is an invalid character for an integer. Some examples:

```
var iNum1 = parseInt("1234blue");     //returns 1234
var iNum2 = parseInt("0xA");          //returns 10
var iNum3 = parseInt("22.5");         //returns 22
var iNum4 = parseInt("blue");         //returns NaN
```

Form Basics

An HTML form is defined by using the <form/> element, which has several attributes:

- **method —** Indicates whether the browser should sent a GET request or a POST request
- **action —** Indicates the URL to which the form should be submitted
- **enctype —** The way the data should be encoded when sent to the server. The default is application/x-www-url-encoded, but it may be set to multipart/form-data if the form is uploading a file.
- **accept —** Lists the mime types the server will handle correctly when a file is uploaded
- **accept-charset —** Lists the character encodings that are accepted by the server when data is submitted

A form can contain any number of input elements:

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

- **<input/> —** The main HTML input element. The type attribute determines what type of input control is displayed:
- • gtext • h • \ A single-line text box
- • g**radio** • h • \ A radio button
- • g**checkbox** • h • \ A check box
- • g**file** • h • \ A file upload text box
- • g**password** • h – A password text box (where characters are not displayed as you type)
- • g**button** • h • \ A generic button that can be used to cause a custom action
- • g**submit** • h • \ A button whose sole purpose is to submit the form
- • g**reset** • h • \ A button whose sole purpose is to reset all fields in the form to their default values
- • g**hidden** • h • \ An input field that isn • ft displayed on screen
- • g**image** • h • \ An image that is used just like a Submit button
- <**select**/> **—** Renders either a combo box or a list box composed of values defined by <option/> elements
- **<textarea/>** • \ Renders a multiline text box in a size determined by the rows and cols attributes.

Here is a simple form using the various input elements:

```
<html><head><title>Sample Form</title></head>
    <body>
        <form method=”post” action=”handlepost.jsp”>
            <!-- regular textbox -->
            <label for=”txtName”>Name:</label><br />
            <input type=”text” id=”txtName” name=”txtName” /><br />
            <!-- password textbox -->
            <label for=”txtPassword”>Password:</label><br />
            <input type=”password” id=”txtPassword” name=”txtPassword” /><br />
            <!-- age comboxbox (drop-down) -->
            <label for=”selAge”>Age:</label><br />
            <select name=”selAge” id=”selAge”>
                <option>18-21</option>
                <option>22-25</option>
                <option>26-29</option>
                <option>30-35</option>
                <option>Over 35</option></select><br />
        <!-- multiline textbox -->
        <label for=”txtComments”>Comments:</label><br />
    <textarea rows=”10” cols=”50”id=”txtComments”name=”txtComments”></textarea><br />
        <!-- submit button -->
        <input type=”submit” value=”Submit Form” />
    </form></body></html>
```

In this example, five form fields are described: a regular text box, a password text box, a combo box, a multiline text box, and a Submit button. Note that with the exception of the Submit button, each field is preceded by a <label/> element. This element is used behind the scenes to logically tie a label to a particular form field. This feature is very useful for screen readers used by visually impaired users. The for attribute indicates the ID of the

form field it identifies. Because of this, each form field should have name and id equal to the same value (name is submitted to the server; id identifies the element on the client).

Each type of form field can be manipulated using JavaScript. The <form/> element itself can also be controlled using JavaScript to provide further control over transmission of data.

Scripting the <form/> Element

Using JavaScript with the <form/> element is different from using other HTML elements. You aren't limited to just using the core DOM methods to access forms; you can access both the <form/> element itself and the form fields in a few different ways. This section covers the basic information you need to begin scripting forms.

Getting form references

Before scripting a form, you first must get a reference to the <form/> element. This can be done in a number of different ways.

First, you can use the typical method of locating an element in a DOM tree, that is, use getElementById() and pass in the form • fs ID: var oForm = document.getElementById(• gform1 • h);

Accessing form fields

Every form field, whether it is a button, text box, or other, is contained in the form • fs elements collection. You can access the various fields in the collection by using their name attributes or their positions in the collection:

```
var oFirstField = oForm.elements[0];   //get the first form field
var oTextbox1 = oForm.elements[ • gtextbox1 • h];//get the field with the name  • gtextbox1 • h
```

In the shorthand version for accessing an element by its name, every form field becomes a property of the form itself and can be accessed directly by using its name:

```
var oTextbox1 = oForm.textbox1; //get the field with the name "textbox1"
```

If the name has a space in it, use bracket notation instead:

```
var oTextbox1 = oForm["text box 1"]; //get the field with the name "text box 1"
```

Of course, you can still use document.getElementById() with a form field • fs ID to retrieve it directly. The methods discussed in this section are most useful when you need to iterate over all the fields in a single form.

Form field commonalities

All form fields (except for hidden fields) contain common properties, methods, and events:

- ❑ The disabled property is used both to indicate whether the control is disabled as well as to actually disable the control (a disabled control doesn • ft allow any user input, but gives no visual indication that the control is disabled).
- ❑ The form property is a pointer back to the form of which the field is a part.
- ❑ The blur() method causes the form field to lose focus (by shifting the focus

elsewhere).

- ❑ The focus() method causes the form field to gain focus (the control is selected for keyboard interaction).
- ❑ The blur event occurs when the field loses focus; the onblur event handler is then executed.
- ❑ The focus event occurs when the field gains focus; the onfocus event handler is then executed.

For example:

```
var oField1 = oForm.elements[0];
var oField2 = oForm.elements[1];
//set the first field to be disabled
oField1.disabled = true;
//set the focus to the second field
oField2.focus();
//is the form property equal to oForm?
alert(oField1.form == oForm);                    //outputs "true"
```

These properties, methods, and events can come in handy when advanced functionality is needed, such as when you want to move the focus to the first field.

Hidden fields only support the form property, but none of the methods or events common to form fields.

Focus on the first field

When a form is displayed on a Web page, the focus is typically not on the first control. It's easy to change this with a generic script that can be used on any form page.

Many developers just put the following in the page • fs onload event handler:

```
document.forms[0].elements[0].focus();
```

This works in most situations, but consider the problem when the first element in the form is a hidden field, an element that doesn • ft support the focus() method. In this case, you • fd be greeted with a JavaScript error. The key is to set the focus to the first visible form field, and you can write a short method for that.

All the methods pertaining to forms in this lecture are written to an object called FormUtil for easy encapsulation: `var FormUtil = new Object;`

The FormUtil object is only intended to keep similar functions grouped together; you may choose to provide these methods in a separate object or on their own.

The method to set the focus on the first field first checks to ensure that a form exists on the page. It does this by checking the value of document.forms.length:

```
FormUtil.focusOnFirst = function () {
        if (document.forms.length > 0) {                    //...                }
};
```

After you know at least one form is present, you can start to iterate through the form fields until you find the first one that isn't hidden.

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

```
FormUtil.focusOnFirst = function () {
    if (document.forms.length > 0) {
        for (var i=0; i < document.forms[0].elements.length; i++) {
            var oField = document.forms[0].elements[i];
            if (oField.type != "hidden") {
                oField.focus();
                return;
            }
        }
    }
};
```

This method can then be called in the onload event handler:

```
<body onload="FormUtil.focusOnFirst()">
```

Be careful when using this method. In slow-loading pages, it is possible that the user may start typing into a field before the page has been fully loaded. When the focus is then set to the first field, it disrupts the user's input. To prepare for this issue, first check for a value in the first field; if one is there, don't set the focus to it.

Submitting forms

In regular HTML, you submit the form by using a Submit button or an image that acts like a Submit button:

```
<input type="submit" value="Submit" />        <!-- submit button -->
<input type="image" src="submit.gif" />       <!-- image button -->
```

When the user clicks either one of these buttons, the form is submitted without requiring any additional coding. If you press Enter on the keyboard when one of these types of buttons is present, the browser submits the form as if the button were clicked.

You can test to see if a form is submitting by providing an alert for the action attribute:

```
<form method="post" action="javascript:alert('Submitted')">
```

This submits the form to the JavaScript function, which just pops up an alert with the word • gSubmitted • h in it. This is helpful to test form submission because it doesn • ft actually involve going back to the server.

If you want to submit the form without using one of the previously mentioned buttons, you can use the submit() method. The submit() method is part of the DOM definition of a <form/> element and can be used anywhere on a page. To use this method, you must first get a reference to the <form/> element either by using getElementById() or by using the document.forms collection. Each of the following three lines is an acceptable way to reference a form:

```
oForm = document.getElementById("form1");
oForm = document.forms["form1"];
oForm = document.forms[0];
```

After getting the form reference, you can just call the submit() method directly:

```
oForm.submit();
```

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

You can actually mimic the behavior of a Submit button by creating a generic button and assigning its onclick event handler to submit the form:

```
<input type="button" value="Submit Form" onclick="document.forms[0].submit()" />
```

Submit only once

A constant problem in Web forms is that users get very impatient when submitting a form. If the form doesn't disappear right away when they click the Submit button, users often click multiple times. The problems this causes vary from creating duplicate requests to charging a credit card more than once. The solution is a very simple one: After the user clicks the Submit button, you disable it. Here's how: Use a regular button, not a Submit button, and disable the button after the user clicks it. So instead of using this code: `<input type="submit" value="Submit" />`

use this code: `<input type="button" value="Submit" onclick="this.disabled=true; this.form.submit()" />`

When this button is clicked, it is disabled by setting the disabled property to true. Then, the form is submitted (note that the code uses the this keyword to reference the button and the form property to reference the form that it's a part of). This code can also be encapsulated in a function, if you so desire.

Resetting forms

If you want to provide the user a way to reset all form fields to their default values, you can use an HTML Reset button: `<input type="reset" value="Reset Values" />`

Similar to a Submit button, a Reset button requires no scripting for the browser to know what to do when it is clicked. Also similar to the Submit button, a reset event fires when the button is clicked: `<form method="post" action="javascript:alert('Submitted') " onreset="alert('I am resetting') ">`

Of course, you can also use the onreset event handler to cancel the form reset. The form does have a reset() method that can reset the form directly from script without using a Reset button:

```
<input type="button" value="Reset" onclick="document.forms[0].reset()" />
```

Unlike submit(), using reset() still fires the reset event and the onreset event handler is still executed.

Validating dates

For many Web developers, dates are a major headache. Despite the advent of nifty layer-based, pop-up calendar systems, users really just want to be able to type in a date. Most developers cringe at the idea of letting a user manually enter a date. Many different date patterns are used around the world, not to mention the internationalized month and day names! Many sites use three form fields for date entry, usually comprised of two combo boxes (one with month names, the other with day numbers) and a text field for the year (although sometimes this, too, is a combo box). Although this approach is okay, it still leaves users wanting to type in a date, which is much faster than tabbing through

three fields and clicking up or down to select an item in a combo box.

As a quick review, these are the supported patterns:

- m/d/yyyy (such as 6/13/2004)
- mmmm d, yyyy (such as January 12, 2004)

It's much easier to use a function to check if a date is valid, so you can wrap the regular expression and the test in a function called isValidDate():

```
function isValidDate(sText) {
    var reDate = /(?:0[1-9]|[12][0-9]|3[01])\/(?:0[1-9]|1[0-2])\/(?:19|20\d{2})/;
    return reDate.test(sText);
}
```

The isValidDate() function is then called like this:

```
alert(isValidDate("5/5/2004"));          //outputs "true"
alert(isValidDate("10/12/2009"));            //outputs "true"
alert(isValidDate("6/13/2000"));         //outputs "false"
```