Web Design Course

Advance Java Script

Lecture by Mis. Zon and Mr. Kaung Sett

IU #	IU Description	Required / Optional
01	Web Design Concepts	Required
02	HTML Basics	Required
03	Advanced HTML & Web Browsers	Required
04	Structuring & Styling with CSS	Required
05	Working with CSS : An Example	Required
06	Javascript Basics	Required
07	**Advanced Javascript**	**Required**

IU Contents

S. No.	Topic Description	Required / Optional
01	JavaScript Function	
02	Browser Object Model	
03	Document Object Model	
04	JavaScript Object Notation	
05	DOM Methods and Events	
06	Form Validation with JavaScript	
07	Ajax	
08	JSON	

- Functions are **defined** with the **function** keyword.
- Can use a function **declaration** or a function **expression**.

Function Declarations:

Syntax:

```
function functionName(parameters)
    {
            //code to be executed
    }
```

Example:

```
function multiplication(a, b)
{
   return a * b;
}
```

❑ Function Expressions

- A JavaScript function can also be defined using an **expression**.
- A function expression can be stored in a variable.

Example:

```
var y = function (p, q) {return p * q};
```

The Function() Constructor

- In the previous examples, JavaScript functions are defined with the **function** keyword.
- Functions can also be defined with a built-in JavaScript function constructor called Function().

Example:

```
var functionABC = new Function("a", "b", "return a * b");

var x = functionABC(3, 3);
```

❑ Function Hoisting

- Hoisting is JavaScript's default behavior of moving **declarations** to the top of the current scope.
- Hoisting applies to variable declarations and to function declarations.
- Because of this, JavaScript functions can be called before they are declared

Example:

```
functionXXX(5);

function functionXXX(a)
{
   return a * a;
}
```

❑ Self-Invoking Functions

- Function expressions can be made "self-invoking".
- A self-invoking expression is invoked (started) automatically, without being called.
- Function expressions will execute automatically if the expression is followed by ().
- You cannot self-invoke a function declaration.
- You have to add parentheses around the function to indicate that it is a function expression.

Example:

```
(function () {
        var y = "Hello world!!";      // invoke myself
    })();
```

❑ Functions Can Be Used as Values

- JavaScript functions can be used as values:

Example:

```
function functionXXX(p, q)
 {
    return p * q;
}

var y = functionXXX(3, 3);
```

❑ Functions are Objects

- The **typeof** operator in JavaScript returns "function" forfunctions.
- But, JavaScript functions can best be described as objects.
- JavaScript functions have both **properties** and **methods**.
- The arguments.length property returns the number of arguments received when the function was invoked:
- Also you can get the number of paraemeters in the function using <functionname>.length

Example:

```
function functionXXX(a, b)
{
   alert(functionXXX.length)
   return arguments.length;
}
```

JavaScript Function Parameters

- A JavaScript function does not perform any checking on parameter values (arguments).
- Function Parameters and Arguments

```
functionName(parameter1, parameter2, parameter3)
 {
                code to be executed
}
```

❑ JavaScript Function Invocation

- The code in a function is not executed when the function is **defined**. It is executed when the function is **invoked**.
- Some people use the term "**call a function**" instead of "**invoke a function**".
- It is also quite common to say "call upon a function", "start a function", or "execute a function".

Example:

```
function functionXXX(a, b)
{
   return a * b;
}
myFunction(2, 20);
```

❑ Invoking a Function as a Method

- In JavaScript you can define function as object methods.
- The following example creates an object (**myObject**), with two properties (**firstName** and **lastName**), and a method (**fullName**):

Example:

```
var Object ={
                fName:“ABC",
                lName: “PQR",
                fullName: function (){
                   return this.fName + " " + this.lName;
                }
            }
            Object.fullName();        // Will return “ABC PQR"
```

❑ Invoking a Function with a Function Constructor

- If a function invocation is preceded with the **new** keyword, it is a constructor invocation.
- It looks like you create a new function, but since JavaScript functions are objects you actually create a new object:

Example:

```
// This is a function constructor:
function func(arg1, arg2)
{
  this.fName = arg1;
  this.lName = arg2;
} // This creates a new object
var y = new func("ABC","XYZ");
alert(y.fName);                          // Will return “ABC"
```

❑ Invoking a Function with a Function Method

- In JavaScript, functions are objects. JavaScript functions have properties and methods.
- **call()** and **apply()** are predefined JavaScript function methods. Both methods can be used to invoke a function, and both methods must have the owner object (this) as first parameter.
- The apply() method is identical to call(), except apply() requires an array as the second parameter. The array represents the arguments for the target method.

Example:

```
var mObject;
function func(a,b)
{
  return a * b;
}
mObject = func.call(mObject, 10, 3);     // Will return 30
alert(mObject);
```

Browser Object Model

- There are no official standards for the **B**rowser **O**bject **M**odel (BOM).
- Since modern browsers have implemented (almost) the same methods and properties for JavaScript interactivity, it is often referred to, as methods and properties of the BOM.

The Window Object

- The **window** object is supported by all browsers. It represents the browser's window.
- Window object is created with every instance of <body> or <frameset> tag
- All global JavaScript objects, functions, and variables automatically become members of the window object.
- Global variables are properties and methods of the window object.

  window.document.getElementById("header");

  is the same as:

  document.getElementById("header");

Other Window Methods

- window.open() - open a new window
- window.close() - close the current window
- window.moveTo() -move the current window
- window.resizeTo() -resize the current window

Window Screen

- The window.screen object contains information about the user's screen.
- The **window.screen** object can be written without the window prefix. Properties:
  - screen.width
  - screen.height
  - screen.availWidth
  - screen.availHeight
  - screen.colorDepth
  - screen.pixelDepth

Windows Screen Properties

- **Window Screen Width**

The screen.width property returns the width of the visitor's screen in pixels.

- **Window Screen Height**

The screen.height property returns the height of the visitor's screen in pixels.

- **Window Screen Pixel Depth**

The screen.pixelDepth property returns the pixel depth of the screen.

- **Window Screen Available Width**

The screen.availWidth property returns the width of the visitor's screen, in pixels, minus interface features like the Windows Taskbar

- **Window Screen Available Height**

The screen.availHeight property returns the height of the visitor's screen, in pixels, minus interface features like the Windows Taskbar.

JavaScript Window Location

- The window.location object can be used to get the current page address (URL) and to redirect the browser to a new page.
- The **window.location** object can be written without the window prefix.
- **Some examples:**
  - **window.location.href** returns the href (URL) of the current page
  - **window.location.hostname** returns the domain name of the web host
  - **window.location.pathname** returns the path and filename of the current page
  - **window.location.protocol** returns the web protocol used (http:// or https://)
  - **window.location.assign** loads a new document

JavaScript Window History

- The window.history object contains the browsers history.
- The **window.history** object can be written without the window prefix.
- To protect the privacy of the users, there are limitations to how JavaScript can access this object.

Some methods:

- **history.back()**
  - loads the previous URL in the history list.
  - same as clicking back in the browser
- **history.forward()**
  - loads the next URL in the history list.
  - same as clicking forward in the browser

JavaScript Window Navigator

- The window.navigator object provides information about the visitor's browser.
- The **window.navigator** object can be written without the window prefix.

Some examples:

- navigator.appName
- navigator.appCodeName
- navigator.platform

- **Navigator Cookie Enabled -** The property cookieEnabled returns true if cookies are enabled, otherwise false
- **The Browser Names -** The properties **appName** and **appCodeName** return the name of the browser
- **The Browser Engine -** The property **product** returns the engine name of the browser
- **The Browser Version I -**The property **appVersion** returns version information about the browser:

- **The Browser Version II -** The property **userAgent also** returns version information about the browser
- **The Browser Platform -** The property **platform** returns the browser platform (operating system)
- **The Browser Language -** The property **language** returns the browser's language
- **Is Java Enabled? -** The method **javaEnabled()** returns true if Java is enabled.

❑ JavaScript Popup Boxes

- JavaScript has three types of popup boxes: Alert box, Confirm box, and Prompt box.
- **Alert Box**
  - An alert box is often used if you want to make sure information comes through to the user.
  - When an alert box pops up, the user will have to click "OK" to proceed.

Syntax:

```
window.alert("sometext");
```

- **Confirm Box**
  - A confirm box is used if you want the user to verify or accept something.
  - When a confirm box pops up, the user will have to click either "OK" or "Cancel" to proceed.
  - If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

Syntax:

```
window.confirm("sometext");
```

- **Prompt Box**
  - A prompt box is often used if you want the user to input a value before entering a page.
  - When a prompt box pops up, the user will have to click either "OK" or "Cancel" to proceed after entering an input value.
  - If the user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

Syntax:

```
window.prompt("sometext","defaultText");
```

- **Line Breaks**
  - To display line breaks inside a popup box, use a back-slash followed by the character n.

Example:

```
alert("Hello\nWorld?");
```

JavaScript Timing Events

- The window object provides execution of code at specified time intervals.
- These time intervals are called timing events.

The two key methods to use with JavaScript are:

- **setTimeout(*function, milliseconds*)**
  Executes a function, after waiting a specified number of milliseconds.
- **setInterval(*function, milliseconds*)**
  Same as setTimeout(), but repeats the execution of the function continuously.

JavaScript Cookies

- Cookies are data, stored in small text files, on your computer.
- When a web server has sent a web page to a browser, the connection is shut down, and the server forgets everything about the user.
- Cookies were created to solve the problem "how to remember information about the user":
- When a user visits a web page, his name can be stored in a cookie.
- Next time the user visits the page, the cookie "remembers" his name.
- Cookies are saved in name-value pairs

  like: username=Johny Doey

- JavaScript can create, read, and delete cookies with the **document.cookie** property.
- With JavaScript, a cookie can be created like this:

```
document.cookie="username=abc";
```

- You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

```
document.cookie="username=abc; expires=Mon, 14 Mar 2015 12:00:00 UTC";
```

- **Reading & Deleting a Cookie with JavaScript**
  - With JavaScript, cookies can be read likethis:

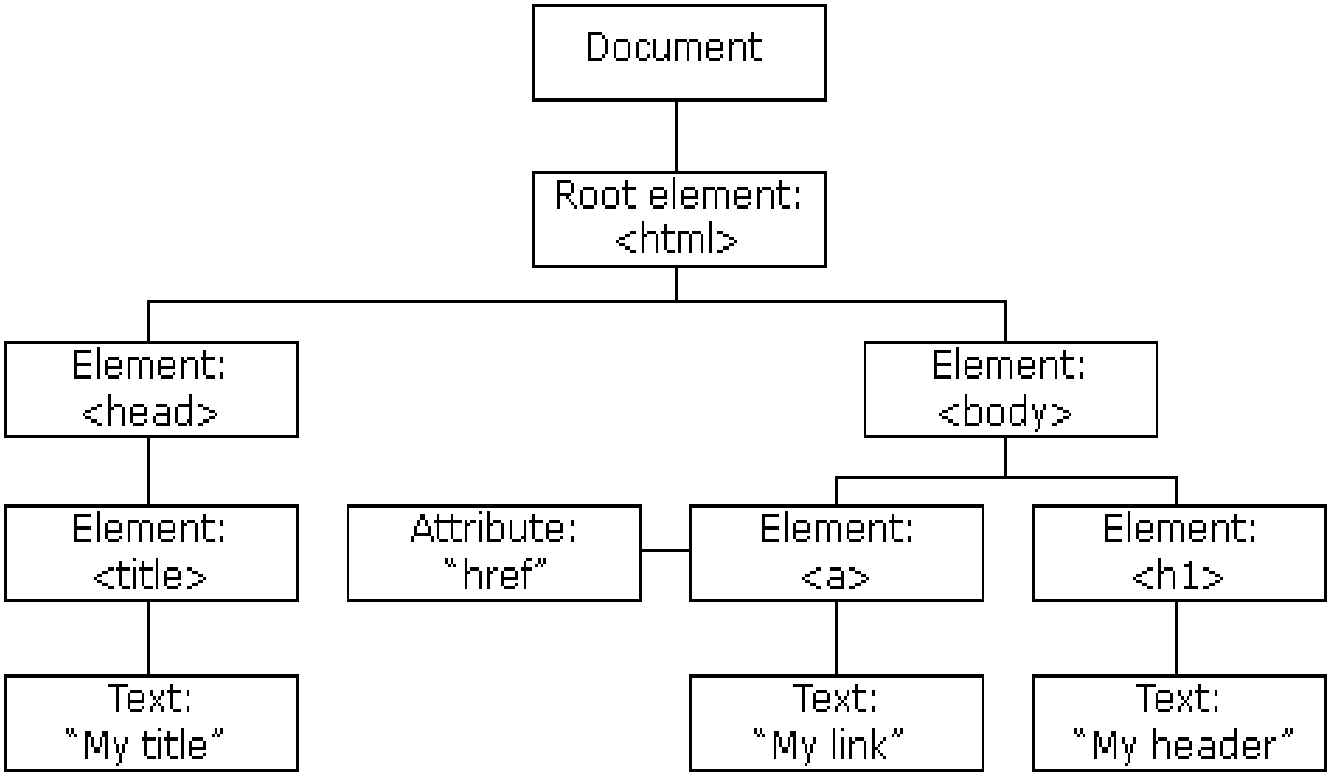
```
var x = document.cookie;
```

  - Deleting a cookie is very simple. Just set the expires parameter to a passed date:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Document Object Model

- With the HTML DOM, JavaScript can access and change all the elements of an HTML document.
- The **HTML DOM** model is constructed as a tree of **Objects**:



JavaScript HTML DOM Document

- The HTML DOM document object is the owner of all other objects in your web page.

Method	Description
document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name
document.getElementsByClassName(*name*)	Find elements by class name

❑ Changing HTML Elements

Method	Description
element.innerHTML = *new html content*	Change the inner HTML of an element
element.attribute = new value	Change the attribute value of an HTML element
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element

❑ Adding Events Handlers

Method	Description
document.getElementById(*id*).onclick = function(){*code*}	Adding event handler code to an onclick event

JavaScript HTML DOM Elements

- Find and access HTML elements in an HTML page.
- Find an HTML Element
  - Finding HTML elements by id (document.getElementById())
  - Finding HTML elements by tag name (document. getElementsByTagName())
  - Finding HTML elements by class name (document. getElementsByClassName())
  - Finding HTML elements by CSS selectors (document. querySelectorAll())
  - Finding HTML elements by HTML object collections (document.forms["frm1"];)

- The querySelectorAll() method returns all elements in the document that matches a specified CSS selector(s), as a static NodeList object.
- The NodeList object represents a collection of nodes. The nodes can be accessed by index numbers. The index starts at 0.
- Get all elements in the document with class="example":

```
var x = document.querySelectorAll(".example");
```

- Get all <p> elements in the document, and set the background color of the first <p> element (index 0):

```
// Get all <p> elements in the document
var x = document.querySelectorAll("p");

// Set the background color of the first <p> element
x[0].style.backgroundColor = "red";
```

JavaScript HTML DOM - Changing HTML

The HTML DOM allows JavaScript to change the content of HTML elements.

- **Changing HTML Content**
  - The easiest way to modify the content of an HTML element is by using the innerHTML property.
  - To change the content of an HTML element, use this syntax:

    document.getElementById(id).innerHTML = new HTML

- **Changing the Value of an Attribute**
  - To change the value of an HTML attribute, use this syntax:

    document.getElementById(id).attribute=new value

JavaScript HTML DOM - Changing CSS

The HTML DOM allows JavaScript to change the style of HTML elements.

- **Changing HTML Style**
  - To change the style of an HTML element, use this syntax:

    document.getElementById(*id*).style.*property=new style*

Using Events

- The HTML DOM allows you to execute code when an event occurs.
- Events are generated by the browser when "things happen" to HTML elements:
  - An element is clicked on
  - The page has loaded
  - Input fields are changed

JavaScript HTML DOM Animation

- **Create an Animation Container**
  - All animations should be relative to a container element.

Example:

```
<div id ="container">
   <div id ="animate">My animation will go here</div>
</div>
```

- **The style Elements**
  - The container element should be created with style = "position: relative".
  - The animation element should be created with style = "position: absolute".

JavaScript HTML DOM Events

- HTML DOM allows JavaScript to react to HTML events
- A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.
- To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:

  onclick=*JavaScript*
- Examples of HTML events:
  - When a user clicks the mouse
  - When a web page has loaded
  - When an image has been loaded
  - When the mouse moves over an element
  - When an input field is changed
  - When an HTML form is submitted
  - When a user strokes a key

- **HTML Event Attributes**

To assign events to HTML elements you can use event attributes.

Example:

Assign an onclick event to a button element:

```
<button id="btn" onclick="printDate()">Click</button>
```

- **Assign Events Using the HTML DOM**

The HTML DOM allows you to assign events to HTML elements using JavaScript:

Example:

Assign an onclick event to a button element:

```
<script>
  document.getElementById("btn").onclick = printDate;
</script>
```

- **The onload and onunload Events**
  - The onload and onunload events are triggered when the user enters or leaves the page.
  - The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.
  - The onload and onunload events can be used to deal with cookies.

Example:

```
<body onload="displayCookies()">
```

- **The onchange Event**

  The onchange event are often used in combination with validation of input fields.

❑ JavaScript HTML DOM EventListener

- **The addEventListener method**

The addEventListener() method attaches an event handler to the specified element.

Syntax:

```
element.addEventListener(event, function, useCapture);
```

Example:

Alert "Hello World!" when the user clicks on an element:

```
element.addEventListener("click", functionXXX);

function functionXXX(){
    alert ("Hello World!");
}
```

- **The removeEventListener() method**

The removeEventListener() method removes event handlers that have been attached with the addEventListener() method

Example:

```
element.removeEventListener("mousemove", functionXXX);
```

❑ JavaScript - HTML DOM Methods

Method	Description
write("string")	writes the given string on the doucment.
writeln("string")	writes the given string on the doucment with newline character at the end.
getElementById()	returns the element having the given id value.
getElementsByName()	returns all the elements having the given name value.
getElementsByTagName()	returns all the elements having the given tag name.
getElementsByClassName()	returns all the elements having the given class name.

❑ HTML/DOM events for JavaScript

Events	Description
onclick	occurs when element is clicked.
ondblclick	occurs when element is double-clicked.
onfocus	occurs when an element gets focus such as button, input, textarea etc.
onblur	occurs when form looses the focus from an element.
onsubmit	occurs when form is submitted.
onmouseover	occurs when mouse is moved over an element.
onmouseout	occurs when mouse is moved out from an element (after moved over).
onmousedown	occurs when mouse button is pressed over an element.
onmouseup	occurs when mouse is released from an element (after mouse is pressed).

Events	Description
onload	occurs when document, object or frameset is loaded.
onunload	occurs when body or frameset is unloaded.
onscroll	occurs when document is scrolled.
onresized	occurs when document is resized.
onreset	occurs when form is reset.
onkeydown	occurs when key is being pressed.
onkeypress	occurs when user presses the key.
onkeyup	occurs when key is released.

JavaScript Objects

- In JavaScript, almost "everything" is an object, except primitive values.
  - Booleans can be objects (or primitive data treated as objects)
  - Numbers can be objects (or primitive data treated as objects)
  - Strings can be objects (or primitive data treated as objects)
  - Dates are always objects
  - Maths are always objects
  - Regular expressions are always objects
  - Arrays are always objects
  - Functions are always objects
  - Objects are objects
- Primitive values are: strings (“ABC"), numbers (5.54), true, false, null, and undefined.

Creating a JavaScript Object

- Can define and create your own objects.
- **There are different ways to create new objects:**
  - Define and create a single object, using an object literal.
  - Define and create a single object, with the keyword new.
  - Define an object constructor, and then create objects of the constructed type.
- **Using an Object Literal**
  - easiest way to create a JavaScript Object.
  - Using an object literal, you both define and create an object in one statement.
  - An object literal is a list of name:value pairs (like age:55) inside curly braces {}.
  - The following example creates a new JavaScript object with four properties: **Example:**

```
var person = {fName:“abc", lName:“pqr", age:55, eyeColor:"black"};
```

- **Using the JavaScript Keyword new**

Example creates a new JavaScript object with four properties:

Example:

```
var person = new Object();
person.fName = “ABC";
person.lName = “PQR";
person.age = 55;
person.eyeColor = "black";
```

- **The *this* Keyword**
  - In JavaScript, the thing called **this**, is the object that "owns" the JavaScript code.
  - The value of **this**, when used in a function, is the object that "owns" the function.
  - The value of **this**, when used in an object, is the object itself.
  - The **this** keyword in an object constructor does not have a value. It is only a substitute for the new object.
  - The value of **this** will become the new object when the constructor is used to create an object

- **Using an Object Constructor**
  - Sometimes we like to have an "object type" that can be used to create many objects of one type.
  - The standard way to create an "object type" is to use an object constructor function:

Example:

```
function person(first, last, age, eye) {
    this.fName = first;
    this
    this
    = ev
}
var m
var m
```

JavaScript Objects are Mutable

- Example:Objects are mutable: They are addressed by reference, not by value.
- If y is an object, the following statement will not create a copy of z: var x = z; // This will not create a copy of z.
- The object x is not a **copy** of z. It **is** z. Both x and z points to the same object.
- Any changes to z will also change x, because x and z are the same object.

```
var person = {firstName:“ABC", lastName:“PQR", age:55, eyeColor:"blue"}
var x = person;
x.age = 10;          // This will change both x.age and person.age
```

- **Validation:** ensuring that form's values are correct
- some types of validation:
  - preventing blank values (email address)
  - ensuring the type of values
    - integer, real number, currency, phone number, Social Security number, postal
  - address, email address, date, credit card number, ...
  - ensuring the format and range of values (ZIP code must be a 5-digit integer)
  - ensuring that values fit together (user types email twice, and the two must match)

- Below form shows various issues in inputted form values.

- Validation can be performed:
  - **Client-Side** (before the form is submitted)
    - can lead to a better user experience, but not secure
  - **Server-Side** (in Server Script code, after the form is submitted)
    - needed for truly secure validation, but slower
  - both
  - best mix of convenience and security, but requires most effort to program

- All forms on a web page are stored in the **document.forms[]** array.
- JavaScript arrays having a starting index of 0, therefore the first form on a page is **document.forms[0]**, the second form is **document.forms[1]**, and so on.
- However, it is usually easier to give the forms names (with the name attribute) and refer to them that way.
- For example, a form named **LoginForm** can be referenced as **document.LoginForm**.
- The major advantage of naming forms is that the forms can be repositioned on the page without affecting the JavaScript.
- Like with other elements, you can also give your forms ids and reference them with **document.getElementById()**.

- Elements within a form are properties of that form and can be referenced as follows:

  document.formName.elementName

- Text fields and passwords have a value property that holds the text value of the field. The following example shows how JavaScript can access user-entered text:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Form Fields</title>
<script type="text/javascript">
        function changeBg(){
                var userName = document.forms[0].userName.value;
                var bgColor = document.colorForm.color.value;

                document.bgColor = bgColor;
                alert(userName + ", the background color is " + bgColor + ".");
        }
</script>
</head>
```

```
<body>
<h1>Change Background Color</h1>
<form name="colorForm">
        Your Name: <input type="text" name="userName"><br>
        Background Color: <input type="text" name="color"><br>
        <input type="button" value="Change Background"
onclick="changeBg();">
</form>
</body>
</html>
```

- When the user clicks on the "Change Background" button, the changeBg() function is called.
- The values entered into the **userName** and **color** fields are stored in variables (**userName** and **bgColor**).
- This form can be referenced as **forms[0]** or **colorForm**. The userName field is referenced as **document.forms[0].userName.value** and the **color** field is referenced as **document.colorForm.color.value**.

- When the user clicks on a submit button, an event occurs that can be caught with the form tag's **onsubmit** event handler.
- Unless JavaScript is used to explicitly cancel the submit event, the form will be submitted.
- The return false; statement explicitly cancels the submit event.

```
<form action="Process.html" onsubmit="return validate(this);">
```

- Notice that we pass the **validate()** function the **this** object.
- In the case above, the **this** object refers to the form object.
- The entire form object is passed to the **validate()** function.

❑ An Example

```
<form method="post" action="Process.html"
                                    onsubmit="return validate(this);">
        Username: <input type="text" name="Username" size="10"><br>
        Password: <input type="password" name="Password" size="10"><br>
        <input type="submit" value="Submit">
        <input type="reset" value="Reset Form">
</form>
<script type="text/javascript">
function validate(form){
        var userName = form.Username.value;
        var password = form.Password.value;
        if (userName.length === 0) {
                alert("You must enter a username."); return false;
        }
        if (password.length === 0) {
                alert("You must enter a password.");return false;
        }
        return true;
}
</script>
```

- One problem is that the validate() function only checks for one problem at a time.
- That is, if it finds an error, it reports it immediately and does not check for additional errors.
- We can consolidate the errors and show them at one go.
- Another problem is that the code is not written in a way that makes it easily reusable.
- We can have standard function to validate common issues.

```
function validate(form){
        var userName = form.Username.value;
        var password = form.Password.value;
        var errors = [];
        if (!checkLength(userName,1,50)) { errors.push("You must enter a username.");}
        if (!checkLength(password,1,20)) { errors.push("You must enter a password.");}
        if (errors.length > 0) { reportErrors(errors); return false;}
        return true;
}
function checkLength(text, min, max){
        min = min || 1;
        max = max || 10000;
        if (text.length < min || text.length > max) { return false;}
        return true;
}
function reportErrors(errors){
        var msg = "There were some problems...\n";
        var numError;
        for (var i = 0; i<errors.length; i++) {
                numError = i + 1;
                msg += "\n" + numError + ". " + errors[i];
        }
        alert(msg);
}
</script>
```

❑ Form with Radio Buttons

```
<form method="post" action="Process.html"
                                    onsubmit="return validate(this);">
        <strong>Cup or Cone?</strong>
        <input type="radio" name="container" value="cup">Cup
        <input type="radio" name="container" value="plaincone"> Plain cone
        <input type="radio" name="container" value="sugarcone"> Sugar cone
        <input type="radio" name="container" value="wafflecone"> Waffle cone
        <br><br>
        <input type="submit" value="Place Order">
</form>

<script type="text/javascript">
function checkRadioArray(radioButtons){
        for (var i=0; i < radioButtons.length; i++) {
                if (radioButtons[i].checked) {
                        return true;
                }
        }
        return false;
}
```

The **checkRadioArray()** function takes a radio button array as an argument, loops through each radio button in the array, and returns true as soon as it finds one that is checked.

```
function validate(form){
        var errors = [];
        if ( !checkRadioArray(form.container) ) { errors[errors.length] = "You must choose a
cup or cone."; }
        if (errors.length > 0) { reportErrors(errors); return false; }
        return true;
}

function reportErrors(errors){
        var msg = "There were some problems...\n";
        var numError;
        for (var i = 0; i<errors.length;i++) {
                numError = i + 1;
                msg += "\n" + numError + ". " + errors[i];
        }
        alert(msg);
}
</script>
```

❑ Form with a Checkbox

```
<form method="post" action="Process.html" onsubmit="return
validate(this);">
        <input type="checkbox" name="terms">
        I understand that I'm really not going to get any ice cream.
        <br><br>
        <input type="submit" value="Place Order">
</form>

function validate(form){
        var errors = [];
        if ( !checkCheckBox(form.terms) ) {errors[errors.length] = "You must agree to
the terms.";}
        if (errors.length > 0) {reportErrors(errors); return false;}
        return true;
}
function checkCheckBox(cb){
        return cb.checked;
}
```

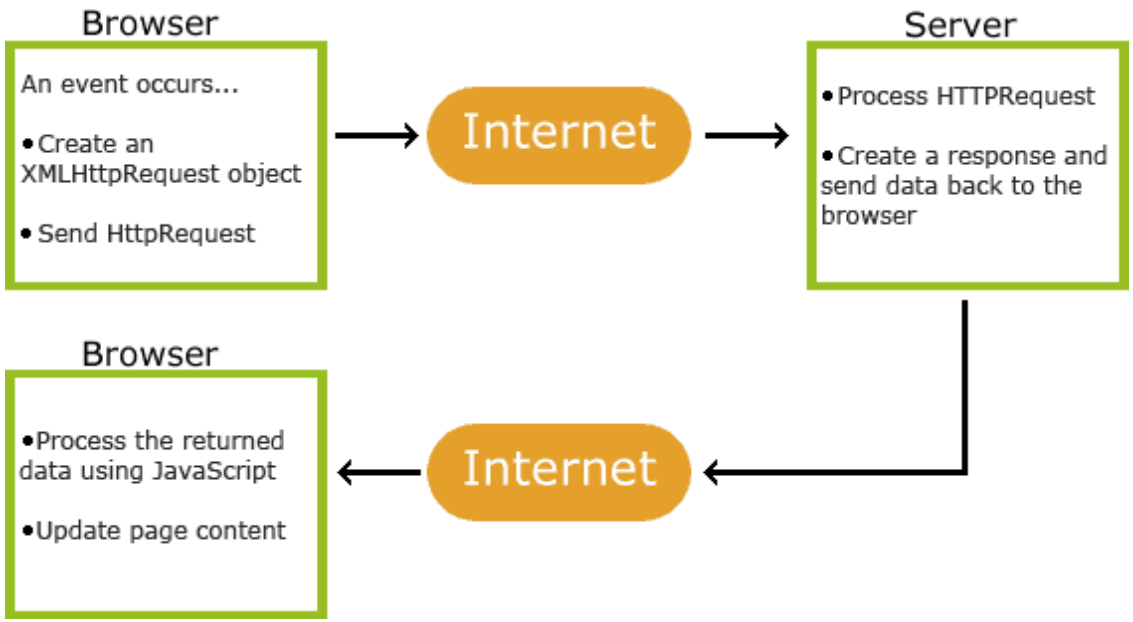
- Select menus contain an array of options.
- The **selectedIndex** property of a select menu contains the index of the option that is selected.
- Often the first option of a select menu is something meaningless like "Please choose an option..."
- The **checkSelect()** function makes sure that the first option is not selected.

```
<form method="post" action="Process.html" onsubmit="return validate(this);">
        <strong>Flavor:</strong>
        <select name="flavor">
                <option value="0" selected></option>
                <option value="choc">Chocolate</option>
                <option value="straw">Strawberry</option>
                <option value="van">Vanilla</option>
        </select>
        <br><br>
        <input type="submit" value="Place Order">
</form>
```

```
function validate(form){
        var errors = [];
        if ( !checkSelect(form.flavor) ) {errors[errors.length] = "You must choose a flavor.";}
        if (errors.length > 0) {reportErrors(errors); return false;}
        return true;
}
function checkSelect(select){
        return (select.selectedIndex >0);
}
```

- You can validate TextArea similar to textbox control

- Ajax stands for Asynchronous JavaScript and Xml
- Update a web page without reloading the page
- Ajax makes web pages more interactive & faster
- Request data from a server - after the page has loaded
- Receive data from a server - after the page has loaded
- Send data to a server - in the background



```
<!DOCTYPE html>
<html>
<body>
<div id="demo">
  <h2>Let AJAX change this  text</h2>
  <button type="button" onclick="loadDoc()">Change Content</button>
</div>
</body>
</html>
```

```
 function loadDoc() {
   var xhttp = new XMLHttpRequest();
   xhttp.onreadystatechange = function() {
     if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML = this.responseText;
     }
   };
   xhttp.open("GET", "ajax_file.html", true);
   xhttp.send();
 }
```

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(*method,url,async,user,psw*)	Specifies the request *method*: the request type GET or POST *url*: the file location *async*: true (asynchronous) or false (synchronous) *user*: optional user name *psw*: optional password
send()	Sends the request to the server Used for GET requests
send(*string*)	Sends the request to the server. Used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number of a request 200: "OK" 403: "Forbidden" 404: "Not Found" For a complete list go to the Http Messages Reference
statusText	Returns the status-text (e.g. "OK" or "Not Found")

THANK YOU