

JavaScript

Notes by rajeeshvengalapurath@gmail.com

Introduction

JavaScript is a single threaded programming language

Callback

- Many actions in JavaScript are **asynchronous**. In other words, we initiate them now, but they finish later
- For instance, we can schedule such actions using `setTimeout`
- Simply put: A callback is a function that is to be executed after another function has finished executing — hence the name 'call back'.
- More complexly put: In JavaScript, functions are objects. Because of this, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called higher-order functions. Any function that is passed as an argument is called a callback function.
- What the heck is the event loop anyway? | Philip Roberts | JSConf EU - <https://www.youtube.com/watch?v=8aGhZQkoFbQ>

```
loadScript('/my/script.js'); //The script is executed "asynchronously", as it
starts loading now, but runs later, when the function has already finished.
// the code below loadScript doesn't wait for the script loading to finish
```

Sample

```
first(callback)
{
    console.log("first started");
    setTimeout(() => {
        callback() //callback is called after 5 seconds
    }, 5000);
    console.log("first ended");
}

//call
first(()=>{
    console.log("callback called");
})
```

Numbers are 64bit

Different ways of defining functions

Defining a function using function declaration

Example 1: Declaring a function first and then calling it.

```
function addNumbers(firstNumber, secondNumber)
{
    var result = firstNumber + secondNumber;
    return result;
}

var sum = addNumbers(10, 20);
document.write(sum);
```

Output : 30

Example 2: A function call is present before the respective function declaration

In Example 1, we are first defining the function and then calling it. The call to a JavaScript function can be present anywhere, even before the function is declared. The following code also works just fine. In the example below, we are calling the function before it is declared.

```
var sum = addNumbers(10, 20);
document.write(sum);

function addNumbers(firstNumber, secondNumber)
{
    var result = firstNumber + secondNumber;
    return result;
}
```

Function Hoisting

By default, JavaScript moves all the function declarations to the top of the current scope. This is called function hoisting. This is the reason JavaScript functions can be called before they are declared.

Defining a JavaScript function using a function expression

A Function Expression allows us to define a function using an expression (typically by assigning it to a variable). There are 3 different ways of defining a function using a function expression.

Anonymous function expression example

In this example, we are creating a function without a name and assigning it to variable add. We use the name of the variable to invoke the function.

```
var add = function (firstNumber, secondNumber)
{
    var result = firstNumber + secondNumber;
    return result;
}
```

```
}

var sum = add(10, 20);
document.write(sum);
```

Functions defined using a function expression are not hoisted. So, this means a function defined using a function expression can only be called after it has been defined while a function defined using standard function declaration can be called both before and after it is defined.

// add() is undefined at this stage

```
var sum = add(10, 20);
document.write(sum);

var add = function (firstNumber, secondNumber)
{
    var result = firstNumber + secondNumber;
    return result;
}
```

Named function expression example

This is similar to the example above. The difference is instead of assigning the variable to an anonymous function, we're assigning it to a named function (with the name `computeFactorial`).

```
var factorial = function computeFactorial(number)
{
    if (number [= 1])
    {
        return 1;
    }

    return number * computeFactorial(number - 1);
}

var result = factorial(5);
document.write(result);
```

The name of the function (i.e `computeFactorial`) is available only within the same function. This syntax is useful for creating recursive functions. If you use `computeFactorial()` method outside of the function it raises 'computeFactorial' is undefined error.

```
var factorial = function computeFactorial(number)
{
    if (number [= 1])
    {
        return 1;
    }

    return number * computeFactorial(number - 1);
}
```

```
var result = computeFactorial(5);
document.write(result);
```

Output : Error - 'computeFactorial' is undefined.

Self invoking function expression example :

```
var result = (function computeFactorial(number)
{
    if (number [= 1])
    {
        return 1;
    }

    return number * computeFactorial(number - 1);
})(5);

document.write(result);
```

Output : 120

These are called with different names

Immediately-Invoked Function Expression (IIFE)

Self-executing anonymous functions

Self-invoked anonymous functions

Closures

What is a closure

A closure is an inner function that has access to the outer function's variables in addition to its own variables and global variables. The inner function has access not only to the outer function's variables, but also to the outer function's parameters. You create a closure by adding a function inside another function.

JavaScript Closure Example

```
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";
    // This inner function has access to the outer function's variables &
    // parameters
    function add()
    {
        return returnValue + (firstNumber + secondNumber);
    }
    return add();
}

var result = addNumbers(10, 20);
document.write(result);
```

Output : Result is : 30

The following code Returns the inner function expression

```
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";
    function add()
    {
        return returnValue + (firstNumber + secondNumber);
    }
    // We removed the parentheses. This will return the inner function expression
    without executing it.
    return add;
}

// addFunc will contain add() function (inner function) expression.
var addFunc = addNumbers(10, 20);
// call the addFunc() function and store the return value in result variable
var result = addFunc();
```

```
document.write(result);
```

Returning and executing the inner function

```
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";
    function add()
    {
        return returnValue + (firstNumber + secondNumber);
    }
    // We removed the parentheses. This will return the inner function add()
    expression without executing it.
    return add;
}

// This returns add() function (inner function) definition and executes it. Notice
the additional parentheses.
var result = addNumbers(10, 20)();
```

```
document.write(result);
```

===JavaScript closure example

Every time we click a button on a web page, we want to increment the click count by 1. There are several ways we can do this.

Using a global variable and incrementing it everytime we click the button : The problem with this approach is that, since clickCount is a global variable any script on the page can accidentally change the variable value.

```
<script type="text/javascript">
    var clickCount = 0;
```

```
</script>
<input type="button" value="Click Me" onclick="alert(++clickCount);" />
==Using a local variable with in a function and incrementing it by calling the
function
The problem with this approach is that, click count is not incremented beyond 1, no
matter how many times you click the button.
```

```
<script type="text/javascript">
    function incrementClickCount()
    {
        var clickCount = 0;
        return ++clickCount;
    }
</script>
<input type="button" value="Click Me" onclick="alert(incrementClickCount());" />
```

Using a JavaScript closure

A closure is an inner function that has access to the outer function's variables in addition to it's own variables and global variables. In simple terms a closure is function inside a function. These functions, that is the inner and outer functions could be named functions or anonymous functions. In the example below we have an anonymous function inside another anonymous function. The variable incrementClickCount is assigned the return value of the self invoking anonymous function.

```
<script type="text/javascript">
    var incrementClickCount = (function ()
    {
        var clickCount = 0;
        return function ()
        {
            return ++clickCount;
        }
    }) ();
</script>
<input type="button" value="Click Me" onclick="alert(incrementClickCount);" />
```

In the example above, in the alert function we are calling the variable incrementClickCount without parentheses. At this point, when you click the button, you get the inner anonymous function expression in the alert. The point I want to prove here is that, the outer self-invoking anonymous function run only once and sets clickCount variable to ZERO, and returns the inner function expression. Inner function has access to clickCount variable. Now every time we click the button, the inner function increments the clickCount variable. The important point to keep in mind is that no other script on the page has access to clickCount variable. The only way to change the clickCount variable is thru incrementClickCount function.

```
<script type="text/javascript">
    var incrementClickCount = (function ()
    {
        var clickCount = 0;
        return function ()
        {
```

```

        return ++clickCount;
    }
    }) ();
</script>
<input type="button" value="Click Me" onclick="alert(incrementClickCount());" />

```

JavaScript closure example

Every time we click a button on a web page, we want to increment the click count by 1. There are several ways we can do this.

Using a global variable and incrementing it everytime we click the button : The problem with this approach is that, since clickCount is a global variable any script on the page can accidentally change the variable value.

```

<script type="text/javascript">
    var clickCount = 0;
</script>
<input type="button" value="Click Me" onclick="alert(++clickCount);" />

```

Using a local variable with in a function and incrementing it by calling the function

The problem with this approach is that, click count is not incremented beyond 1, no matter how many times you click the button.

```

<script type="text/javascript">
    function incrementClickCount()
    {
        var clickCount = 0;
        return ++clickCount;
    }
</script>
<input type="button" value="Click Me" onclick="alert(incrementClickCount());" />

```

Using a JavaScript closure

A closure is an inner function that has access to the outer function's variables in addition to it's own variables and global variables. In simple terms a closure is function inside a function. These functions, that is the inner and outer functions could be named functions or anonymous functions. In the example below we have an anonymous function inside another anonymous function. The variable incrementClickCount is assigned the return value of the self invoking anonymous function.

```

<script type="text/javascript">
    var incrementClickCount = (function ()
    {
        var clickCount = 0;
        return function ()
        {
            return ++clickCount;
        }
    }) ();
</script>

```

```
<input type="button" value="Click Me" onclick="alert(incrementClickCount);" />
```

In the example above, in the alert function we are calling the variable `incrementClickCount` without parentheses. At this point, when you click the button, you get the inner anonymous function expression in the alert. The point I want to prove here is that, the outer self-invoking anonymous function run only once and sets `clickCount` variable to ZERO, and returns the inner function expression. Inner function has access to `clickCount` variable. Now every time we click the button, the inner function increments the `clickCount` variable. The important point to keep in mind is that no other script on the page has access to `clickCount` variable. The only way to change the `clickCount` variable is thru `incrementClickCount` function.

```
<script type="text/javascript">
    var incrementClickCount = (function ()
    {
        var clickCount = 0;
        return function ()
        {
            return ++clickCount;
        }
    })();
</script>
<input type="button" value="Click Me" onclick="alert(incrementClickCount());" />
```

Arguments Object

Arguments object is a local variable available within all functions. It contains all the function parameters that are passed to the function and can be indexed like an array. The `length` property of the arguments object returns the number of arguments passed to the function.

JavaScript arguments object example

```
function printArguments()
{
    document.write("Number of arguments = " + arguments.length + "[br/]")
    for (var i = 0; i < arguments.length; i++)
    {
        document.write("Argument " + i + " = " + arguments[i] + "[br/]");
    }
    document.write("[br/]");
}

printArguments();
printArguments("A", "B");
printArguments(10, 20, 30);
```

Is it possible to pass variable number of arguments to a JavaScript function

Yes, you can pass as many arguments as you want to any javascript function. All the parameters will be stored in the arguments object.

```
function addNumbers()
{
    var sum = 0;
```

```

    document.write("Count of numbers = " + arguments.length + "[br/]")
    for (var i = 0; i < arguments.length; i++)
    {
        sum = sum + arguments[i];
    }
    document.write("Sum of numbers = " + sum);
    document.write("[br/][br/]");
}

addNumbers();
addNumbers(10, 20, 30);

```

The arguments object is available only inside a function body. Attempting to access the arguments object outside a function results in 'arguments' is undefined error. Though you can index the arguments object like an array, it is not an array. It does not have any Array properties except length. For example it does not have the sort() method, that the array object has. However, you can convert the arguments object to an array.

Converting JavaScript arguments object to an array

```

function numbers()
{
    var argsArray = Array.prototype.slice.call(arguments);
    argsArray.sort();
    document.write(argsArray);
}

numbers(50, 20, 40);

```

Output : 20, 40, 50

Converting JavaScript arguments object to an array using array literals

```

function numbers()
{
    var argsArray = [].slice.call(arguments);
    argsArray.sort();
    document.write(argsArray);
}

numbers(50, 20, 40);

```

Output : 20, 40, 50

Error Handling

Use try/catch/finally to handle runtime errors in JavaScript. These runtime errors are called exceptions. An exception can occur for a variety of reasons. For example, referencing a variable or a method that is not defined can cause an exception.

The JavaScript statements that can possibly cause exceptions should be wrapped inside a try block. When a specific line in the try block causes an exception, the control is immediately transferred to the catch block skipping the rest of the code in the try block.

JavaScript try catch example

```
try
{
    // Referencing a function that does not exist cause an exception
    document.write(sayHello());
    // Since the above line causes an exception, the following line will not be
    executed
    document.write("This line will not be executed");
}
// When an exception occurs, the control is transferred to the catch block
catch (e)
{
    document.write("Description = " + e.description + "[br/]");
    document.write("Message = " + e.message + "[br/]");
    document.write("Stack = " + e.stack + "[br/][br/]");
}
document.write("This line will be executed");
```

Please note : A try block should be followed by a catch block or finally block or both.

finally block is guaranteed to execute irrespective of whether there is an exception or not. It is generally used to clean and free resources that the script was holding onto during the program execution. For example, if your script in the try block has opened a file for processing, and if there is an exception, the finally block can be used to close the file.

JavaScript try catch finally example

```
try
{
    // Referencing a function that does not exist cause an exception
    document.write(sayHello());
    // Since the above line causes an exception, the following line will not be
    executed
    document.write("This line will not be executed");
}
// When an exception occurs, the control is transferred to the catch block
catch (e)
{
    document.write("Description = " + e.description + "[br/]");
    document.write("Message = " + e.message + "[br/]");
    document.write("Stack = " + e.stack + "[br/][br/]");
}
finally
{
    document.write("This line is guaranteed to execute");
}
```

Syntax errors and exceptions in JavaScript

try/catch/finally block can catch exceptions but not syntax errors.

Example : In the example, below we have a syntax error - missing the closing parentheses. The associated catch block will not catch the syntax errors.

```
try
{
    alert("Hello";
}
catch (e)
{
    document.write("JavaScript syntax errors cannot be caught in the catch block");
}
```

JavaScript throw statement : Use the throw statement to raise a customized exceptions.

JavaScript throw exception example

```
function divide()
{
    var numerator = Number(prompt("Enter numerator"));
    var denominator = Number(prompt("Enter denominator"));

    try
    {
        if (denominator == 0)
        {
            throw {
                error: "Divide by zero error",
                message: "Denominator cannot be zero"
            };
        }
        else
        {
            alert("Result = " + (numerator / denominator));
        }
    }
    catch (e)
    {
        document.write(e.error + "[br/]");
        document.write(e.message + "[br/]");
    }
}

divide();
```

window onerror event

In general we use try/catch statement to catch errors in JavaScript. If an error is raised by a statement that is not inside a try...catch block, the onerror event is fired.

Assign a function to window.onerror event that you want to be executed when an error is raised as shown below. The function that is associated as the event handler for the onerror event has three parameters:

message Specifies the error message.

URL Specifies the location of the file where the error occurred.

line Specifies the line number where the error occurred.

JavaScript window onerror event example

```
window.onerror = function (message, url, line)
{
    alert("Message : " + message + "\nURL : " + url + "\nLine Number : " + line);
    // Return true to suppress the browser error messages (like in older versions of
    Internet Explorer)
    return true;
}
```

```
NonExistingFunction();
```

If the error is handled by a try/catch statement, then the onerror event is not raised. onerror event is raised only when there is an unhandled exception.

```
window.onerror = function (message, url, line)
{
    alert("Message : " + message + "\nURL : " + url + "\nLine Number : " + line);
    // Return true to suppress the browser error messages (like in older versions of
    Internet Explorer)
    return true;
}

try
{
    NonExistingFunction();
}
catch (e)
{
    document.write(e.message);
}
```

Output : 'NonExistingFunction' is undefined

onerror event handler method can also be used with HTML elements : In the example below, since the image is not existing and cannot be found we get "There is a problem loading the image" error.

```
<script type="text/javascript">
    function imageErrorHandler()
    {
        alert("There is a problem loading the image");
    }
</script>
```

```

```

Timing events

In JavaScript a piece of code can be executed at specified time interval. For example, you can call a specific JavaScript function every 1 second. This concept in JavaScript is called timing events.

The global window object has the following 2 methods that allow us to execute a piece of JavaScript code at specified time intervals.

setInterval(func, delay) - Executes a specified function, repeatedly at specified time interval. This method has 2 parameters. The func parameter specifies the name of the function to execute. The delay parameter specifies the time in milliseconds to wait before calling the specified function.

setTimeout(func, delay) - Executes a specified function, after waiting a specified number of milliseconds. This method has 2 parameters. The func parameter specifies the name of the function to execute. The delay parameter specifies the time in milliseconds to wait before calling the specified function. The actual wait time (delay) may be longer.

Let's understand timing events in JavaScript with an example. The following code displays current date and time in the div tag.

```
<div id="timeDiv" ></div>
<script type="text/javascript">
    function getCurrentDateTime() {
        document.getElementById("timeDiv").innerHTML = new Date();
    }

    getCurrentDateTime();
</script>
```

At the moment the time is static. To make the time on the page dynamic modify the script as shown below. Notice that the time is now updated every second. In this example, we are using **setInterval()** method and calling **getCurrentDateTime()** function every 1000 milli-seconds.

```
<div id="timeDiv" ></div>
<script type="text/javascript">
    setInterval(getCurrentDateTime, 1000);

    function getCurrentDateTime() {
        document.getElementById("timeDiv").innerHTML = new Date();
    }
</script>
```

clearInterval(intervalID) - Cancels the repeated execution of the method that was set up using **setInterval()** method. **intervalID** is the identifier of the repeated action you want to

cancel. This ID is returned from `setInterval()` method. The following example demonstrates the use of `clearInterval()` method.

Starting and stopping the clock with button click : In this example, `setInterval()` method returns the `intervalId` which is then passed to `clearInterval()` method. When you click the "Start Clock" button the clock is updated with new time every second, and when you click "Stop Clock" button it stops the clock.

```
<div id="timeDiv" ></div>
<br />
<input type="button" value="Start Clock" onclick="startClock()" />
<input type="button" value="Stop Clock" onclick="stopClock()" />
<script type="text/javascript">
    var intervalId;

    function startClock() {
        intervalId = setInterval(getCurrentDateTime, 1000);
    }

    function stopClock() {
        clearInterval(intervalId);
    }

    function getCurrentDateTime() {
        document.getElementById("timeDiv").innerHTML = new Date();
    }

    getCurrentDateTime();
</script>
```

Now let's look at example of using `setTimeout()` and `clearTimeout()` functions. The syntax and usage of these 2 functions is very similar to `setInterval()` and `clearInterval()`.

Countdown timer example : When we click "Start Timer" button, the value 10 displayed in the textbox must start counting down. When click "Stop Timer" the countdown should stop. When you click "Start Timer" again, it should start counting down from where it stopped and when it reaches ZERO, it should display done in the textbox and function should return.

// timing events in javascript.png

```
<input type="text" value="10" id="txtBox" />
<br /><br />
<input type="button" value="Start Timer" onclick="startTimer('txtBox')" />
<input type="button" value="Stop Timer" onclick="stopTimer()" />
<script type="text/javascript">
    var intervalId;

    function startTimer(controlId)
    {
        var control = document.getElementById(controlId);
        var seconds = control.value;
```

```

        seconds = seconds - 1;
        if (seconds == 0)
        {
            control.value = "Done";
            return;
        }
        else
        {
            control.value = seconds;
        }

        intervalId = setTimeout(function () { startTimer('txtBox'); }, 1000);
    }

    function stopTimer()
    {
        clearTimeout(intervalId);
    }
</script>

```

Image slideshow using JavaScript

(.getAttribute & .setAttribute are used)

When you click "Start Slide Show" button the image slideshow should start and when you click the "Stop Slide Show" button the image slideshow should stop.

For the purpose of this demo we will be using the images that can be found on any windows machine at the following path.

C:\Users\Public\Pictures\Sample Pictures

At the above location, on my machine I have 8 images. Here are the steps to create the image slideshow using JavaScript.

Step 1 : Open Visual Studio and create a new empty asp.net web application project. Name it Demo.

Step 2 : Right click on the Project Name in Solution Explorer in Visual Studio and create a new folder with name = Images.

Step 3 : Copy the 8 images from C:\Users\Public\Pictures\Sample Pictures to Images folder in your project. Change the names of the images to 1.jpg, 2.jpg etc.

Step 4 : Right click on the Project Name in Solution Explorer in Visual Studio and add a new HTML Page. It should automatically add HTMLPage1.htm.

Step 5 : Copy and paste the following HTML and JavaScript code in HTMLPage1.htm page.

```


<br/>

```

```

<input type="button" value="Start Slide Show" onclick="startImageSlideShow()" />
<input type="button" value="Stop Slide Show" onclick="stopImageSlideShow()" />
<script type="text/javascript">
    var intervalId;

    function startImageSlideShow()
    {
        intervalId = setInterval(setImage, 500);
    }

    function stopImageSlideShow()
    {
        clearInterval(intervalId);
    }

    function setImage()
    {
        var imageSrc = document.getElementById("image").getAttribute("src");
        var currentImageNumber = imageSrc.substring(imageSrc.lastIndexOf("/") + 1,
imageSrc.lastIndexOf("/") + 2);
        if (currentImageNumber == 8)
        {
            currentImageNumber = 0;
        }
        document.getElementById("image").setAttribute("src", "/Images/" +
(Number(currentImageNumber) + 1) + ".jpg");
    }
</script>

```

Event

An event is a signal from the browser that something has happened. For example,

1. When a user clicks on an HTML element, click event occurs
2. When a user moves the mouse over an HTML element, mouseover event occurs

When events occur, we can execute JavaScript code or functions in response to those events. To do this we need to associate JavaScript code or functions to the events. The function that executes in response to an event is called event handler.

In JavaScript, there are several ways to associate an event handler to the event

1. Using the attributes of an HTML tag
2. Using DOM object property
3. Using special methods

In this video we will discuss associating event handler methods to events using the attributes of HTML tags.

In the following example, the code to execute in response to onmouseover & onmouseout events is set directly in the HTML markup. The keyword "this" references the current element. In this example "this" references the button control.

```
<input type="button" value="Click me" id="btn"
onmouseover="this.style.background= 'red'; this.style.color = 'yellow'"
onmouseout="this.style.background= 'black'; this.style.color = 'white'" />
```

The above example, can be rewritten as shown below. In this case the code to execute in response to the event is placed inside a function and then the function is associated with the event.

```
<input type="button" value="Click me" id="btn"
onmouseover="changeColorOnMouseOver()"
onmouseout="changeColorOnMouseOut()" />

<script type="text/javascript">
    function changeColorOnMouseOver()
    {
        var control = document.getElementById("btn");
        control.style.background = 'red';
        control.style.color = 'yellow';
    }

    function changeColorOnMouseOut()
    {
        var control = document.getElementById("btn");
        control.style.background = 'black';
        control.style.color = 'white';
    }
</script>
```

Events are very useful in real-world applications. For example they can be used to

1. Display confirmation dialog box on submitting a form
2. Form data validation and many more

How to show confirmation dialog in JavaScript

```
<input type="submit" value="Submit" id="btn" onclick="return confirmSubmit()" />
<script type="text/javascript">
    function confirmSubmit()
    {
        if (confirm("Are you sure you want to submit"))
        {
            alert("You selected OK");
            return true;
        }
        else
        {
            return false;
            confirm("You selected cancel");
        }
    }
</script>
```

JavaScript form validation example : In this example, both First Name and Last Name fields are required fields. When you type the first character in any of the textbox, the background colour is automatically changed to green. If you delete all the characters you typed or if you leave the textbox without entering any characters the background colour changes to red indicating the field is required. We made this possible using onkeyup and onblur events.

onkeyup occurs when the user releases a key.

onblur occurs when an element loses focus.

```
<table>
  <tr>
    <td>
      First Name
    </td>
    <td>
      <input type="text" id="txtFirstName"
onkeyup="validateRequiredField('txtFirstName')"
      onblur="validateRequiredField('txtFirstName')"/>
    </td>
  </tr>
  <tr>
    <td>
      Last Name
    </td>
    <td>
      <input type="text" id="txtLastName"
onkeyup="validateRequiredField('txtLastName')"
      onblur="validateRequiredField('txtLastName')"/>
    </td>
  </tr>
</table>
<script type="text/javascript">
  function validateRequiredField(controlId)
  {
    var control = document.getElementById(controlId);
    control.style.color = 'white';
    if (control.value == "")
    {
      control.style.background = 'red';
    }
    else
    {
      control.style.background = 'green';
    }
  }
</script>
```

Assigning event handlers in JavaScript using DOM object property

In JavaScript there are several ways to associate an event handler to the event. In Part 36, we discussed, associating event handler methods to events using the attributes of HTML

tags. In this video we will discuss using DOM object property to assign event handlers to events.

First let us understand, what is DOM

DOM stands for Document Object Model. When a browser loads a web page, the browser creates a Document Object Model of that page. The HTML DOM is created as a tree of Objects.

Example :

```
<html>
  <head>
    <title>My Page Title</title>
  </head>
  <body>
    <script type="text/javascript">
    </script>
    <div>
      <h1>This is browser DOM</h1>
    </div>
  </body>
</html>
```

JavaScript can be used to access and modify these DOM objects and their properties. For example, you can add, modify and remove HTML elements and their attributes. Along the same lines, you can use DOM object properties to assign event handlers to events. We will discuss the DOM object in detail in a later video session.

We will continue with the same examples that we worked with in Part 36. Notice that in this case, we are assigning event handlers using the DOM object properties (onmouseover & onmouseout) instead of using the attributes of the HTML tag. We are using this keyword to reference the current HTML element. In this example "this" references the button control.

```
<input type="button" value="Click me" id="btn"/>
<script type="text/javascript">
  document.getElementById("btn").onmouseover = changeColorOnMouseOver;
  document.getElementById("btn").onmouseout = changeColorOnMouseOut;

  function changeColorOnMouseOver()
  {
    this.style.background = 'red';
    this.style.color = 'yellow';
  }

  function changeColorOnMouseOut()
  {
    this.style.background = 'black';
    this.style.color = 'white';
  }
</script>
```

The following example is same as the above. In this case we are assigning an anonymous function to onmouseover & onmouseout properties.

```
<input type="button" value="Click me" id="btn" />
<script type="text/javascript">
    document.getElementById("btn").onmouseover = function ()
    {
        this.style.background = 'red';
        this.style.color = 'yellow';
    }

    document.getElementById("btn").onmouseout = function ()
    {
        this.style.background = 'black';
        this.style.color = 'white';
    }
</script>
```

If an event handler is assigned using both, i.e an HTML attribute and DOM object property, the handler that is assigned using the DOM object property overwrites the one assigned using HTML attribute. Here is an example.

```
<input type="button" value="Click me" id="btn" onclick="clickHandler1()" />
<script type="text/javascript">
    document.getElementById("btn").onclick = clickHandler2;

    function clickHandler1()
    {
        alert("Handler set using HTML attribute");
    }

    function clickHandler2()
    {
        alert("Handler set using DOM object property");
    }
</script>
```

Using this approach you can only assign one event handler method to a given event. The handler that is assigned last wins. In the following example, Handler2() is assigned after Handler1. So Handler2() overwrites Handler1().

```
<input type="button" value="Click me" id="btn" />
<script type="text/javascript">
    document.getElementById("btn").onclick = clickHandler1;
    document.getElementById("btn").onclick = clickHandler2;

    function clickHandler1()
    {
        alert("Handler 1");
    }

    function clickHandler2()
    {
        alert("Handler 2");
    }
</script>
```

EoF