JavaScript Functions Notes (Modified from: w3schools.com)

A function is a block of code that will be executed when "someone" calls it. In JavaScript, we can define our own functions, called ***user-defined*** functions, or we can use ***built-in*** functions already defined in the JavaScript language.

We have already seen examples of built-in functions, such as:

- window.alert( )
- document.write( )
- parseInt( )

Below, we will show how to define our own *user-defined* functions.

JavaScript Function Syntax

A function is written as a code block (inside curly { } braces), preceded by the **function** keyword:

```
function functionname()
{
    some code to be executed
}
```

The code inside the function will be executed when "someone" calls the function.

The function can be called directly when an event occurs (like when a user clicks a button), and it can be called from "anywhere" by JavaScript code.

JavaScript is case sensitive. The function keyword must be written in lowercase letters, and the function must be called with the same capitals as used in the function name.

Place User-Defined Functions in the HEAD Section

JavaScript user-defined functions should be defined in the HEAD section of a HTML document. The reason for placing JavaScript functions in the HEAD section is the function must be defined and loaded into memory before the function can be called by JavaScript code or an event handler.

Example #1 helloAlert()

Below, we define a function named **helloAlert( )** in the HEAD section of the webpage. The function contains one statement of code, to display a message box.

The function is *called* later in a SCRIPT element in the BODY section of the webpage.

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">
    function helloAlert( )
    {
        alert("Hello World!");
    }
</script>
</head>

<body>
<script type="text/javascript">
     helloAlert();
<script>
</body>
</html>
```

define function **helloAlert( )**

call function **helloAlert( )**

Defining Multiple Functions

Multiple JavaScript functions can be defined in the HEAD section of a HTML document.

```
<html>
<head>
<script type="text/javascript">
    function function1( )
    {
        some code
    }
    function function2( )
    {
        some code
    }
</script>
</head>

<body>

…

</body>
</html>
```

define multiple functions in the HEAD section

Calling a Function with Parameters/Arguments

When you call a function, you can pass along some values to it, these values are called *parameters* or *arguments*.

Parameters are the variables we specify when we define the function. When the function is later called somewhere else in the code, arguments are the values passed as parameters.

You can specify as many parameters as you like, separated by commas (,). The parameters then serve as variables that can be used inside the function.

Below is the syntax for defining a function that takes three parameters:

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

Example #2 welcome()

In the following example, we define a function named **welcome( )** that takes in one parameter, named *user*. When the function is called, we pass the argument "John Smith" to the function.

```
<script>
function welcome( user )
{
   alert( "Welcome " + user + " to my website!" );
}
</script>
<script>
   welcome( "John Smith" );
</script>
```

In the function declaration, we specify the **parameter** user

When the function is called, "John Smith" is the **argument** passed to the function

The function above will display the message box "Welcome John Smith to my website!" when the function is called in the webpage.

The function is flexible, you can call the function using different arguments, and different welcome messages will be given:

```
<script>
   welcome( "Sam Davis" );
   welcome( "Alice Johnson" );
</script>
```

The examples above will alert "Welcome Sam Davis to my website!" and "Welcome Alice Johnson to my website!" when the script code is executed.

Functions With a Return Value

Sometimes you want your function to return a value back to where the call was made. This is possible by using the *return* statement. When using the *return* statement, the function will stop executing, and return the specified value.

Example #3 average()

```
1   <html><head>
2   <script>
3       function average( num1, num2, num3 ) {
4           var avg = ( num1 + num2 + num3 ) / 3;
5           return avg;
6       }
7   </script>
8   </head>
9   <body>
10  <script>
11      var x = average( 10 , 20 , 60 );
12  </script>
13  </body></html>
```

The average( ) function above takes three numbers as parameters, and returns the average of the three numbers. On Line 11, when the average( ) function is called, the value **30** is returned and then stored in the variable x.

You can also use the value returned by the average() function without storing it as a variable. For instance, we could call the function and use the return value in the alert( ) function:

```
window.alert(  average( 5,10,30 )  );
```

In the statement above, the average( ) function returns the value **15**, which is then displayed in a message box.

The return statement is also used when you simply want to exit a function. In this case, you do not have to specify any return value.

```
function myFunction( a , b )
{
   if (a > b)
   {
      return;
   }
   x=a+b;
}
```

The function above will exit the function if a > b, and will not calculate the sum of a and b.

Local JavaScript Variables

A variable declared (using var) within a JavaScript function becomes **LOCAL** and can only be accessed from within that function. We say the variable has *local* scope.

In the example below, Line 13 will produce an error, because the variable **avg** is a local variable that only exists within the average( ) function.

```
1   <html><head>
2   <script>
3      function average( num1, num2, num3 ) {
4         var avg = ( num1 + num2 + num3 ) / 3;
5         return avg;
6      }
7   </script>
8   </head>
9   <body>
10  <script>
11     var x = average( 10 , 20 , 60 );
12     document.write( "value of x is: " + x );
13     document.write( "value of avg is: " + avg );
14  </script>
    </body></html>
```

You can have local variables with the same name in different functions, because local variables are only recognized by the function in which they are declared.

Local variables are deleted from memory as soon as the function is completed.

Global JavaScript Variables

Variables declared outside a function, become **GLOBAL**, and all scripts and functions on the web page can access it.

The Lifetime of JavaScript Variables

The *lifetime* of a variable is the period of time when the variable is stored in memory and accessible within a program.

In JavaScript, the lifetime of a variables starts when the variable is declared.

Local variables, or variables declared within a function, are deleted when the function is completed.

Global variables, variables declared outside of a function, are deleted when you close the page.

Calling JavaScript Functions from Event Handlers

Event handlers are JavaScript code that executes when the user performs some action, or triggers an *event*.

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

Event handlers can be placed within the opening tag of most html elements. In each of the following examples, we will use event handlers to call built-in and user-defined JavaScript functions.

The onclick Event

The onclick event is triggered when the user clicks the mouse on a specified element.

Example #1

In the following example, the **onclick** event is specified for a H1 element.
When the user clicks on the H1 element, the onclick event will be fired.

```
<html>
<body>
<h1>Test Page</h1>
<h1 onclick="alert('you clicked me …')" >Click on this text!</h1>
</body>
</html>
```

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 also be used to check whether or not the browser has cookies enabled or disabled.

Example #2

```
<!DOCTYPE html>
<html>
<head>
<script>
function checkCookies()
{
    if ( navigator.cookieEnabled == true )
    {
        alert("Cookies are enabled")
    }
    else
    {
        alert("Cookies are not enabled")
     }
}
</script>
</head>
<body onload="checkCookies()">
<p>An alert box should tell you if your browser has enabled cookies or not.</p>
</body>
</html>
```

The onchange Event

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

Below is an example of how to use the onchange event. The upperCase() function will be called when a user changes the content of an input field.

Example #3

```
<!DOCTYPE html>
<html>
<head>
<script>
function makeUpper()
{
    var text = document.getElementById("fname").value;     // get the value of the fname input box
    var new_text = text.toUpperCase( );                    // convert value to uppercase
    document.getElementById("fname").value = new_text;  // set the value of the fname input box
}
</script>
</head>
<body>
Enter your name:
<input type="text" id="fname" onchange="makeUpper()" >
<p>When you leave the input field, a function is triggered which transforms the input text to upper case.</p>
</body>
</html>
```

In the example above, the **onchange()** event is fired when the user changes the contents of the INPUT element.

The onchange( ) event calls the makeUpper( ) function, which retrieves the value of the INPUT element, converts the value to upper case, then stores the new value in the INPUT element.

In the example above, we use the **document.getElementById( )** method to retrieve a specific html element based on the element's id. Specifically, we retrieve the element with the id equal to fname, then obtain the value of that element.

The onmouseover and onmouseout Events

The onmouseover and onmouseout events can be used to trigger a function when the user mouses over, or out of, an HTML element.

The onmousedown, onmouseup and onclick Events

The onmousedown, onmouseup, and onclick events are all parts of a mouse-click. First when a mouse-button is clicked, the onmousedown event is triggered, then, when the mouse-button is released, the onmouseup event is triggered, finally, when the mouse-click is completed, the onclick event is triggered.