

## JavaScript Basics

In this lesson of the JavaScript tutorial, you will learn...

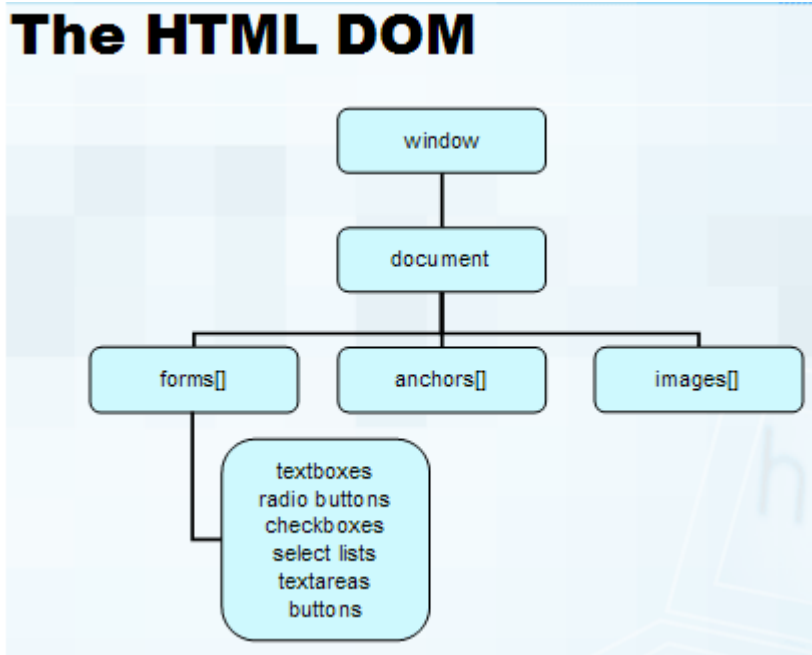
1. To work with the HTML DOM.
2. To follow JavaScript syntax rules.
3. To write JavaScript inline.
4. To write JavaScript in script blocks.
5. To create and link to external JavaScript files.
6. To work with JavaScript objects, methods, and properties.
7. To reference HTML elements with JavaScript.
8. To use event handlers.

## The Name "JavaScript"

In this manual, we refer to the language we are learning as *JavaScript*, which is what it is usually called. However, the name *JavaScript* is owned by Netscape. Microsoft calls its version of the language *JScript*. The generic name of the language is *EcmaScript*.

## The HTML DOM

The HTML Document Object Model (DOM) is the browser's view of an HTML page as an object hierarchy, starting with the browser window itself and moving deeper into the page, including all of the elements on the page and their attributes. Below is a simplified version of the HTML



As shown, the top-level object is window. The document object is a child of window and all the objects (i.e. elements) that appear on the page (e.g. forms, links, images, tables, etc.) are descendants of the document object. These objects can have children of their own. For example,

# JAVA SCRIPT

form objects generally have several child objects, including textboxes, radio buttons, and select menus.

## JavaScript Syntax

### *Basic Rules*

1. JavaScript statements end with semi-colons.
2. JavaScript is case sensitive.
3. JavaScript has two forms of comments:
  - o Single-line comments begin with a double slash (`//`).
  - o Multi-line comments begin with `/*` and end with `*/`.

#### Syntax

```
// This is a single-line comment
```

```
/*  
This is  
a multi-line  
comment.  
*/
```

### *Dot Notation*

In JavaScript, objects can be referenced using dot notation, starting with the highest-level object (i.e, window). Objects can be referred to by name or id or by their position on the page. For example, if there is a form on the page named "loginform", using dot notation you could refer to the form as follows:

#### Syntax

```
window.document.loginform
```

Assuming that loginform is the first form on the page, you could also refer to this way:

#### Syntax

```
window.document.forms[0]
```

A document can have multiple form elements as children. The number in the square brackets ([]) indicates the specific form in question. In programming speak, every document object contains an *array* of forms. The length of the array could be zero (meaning there are no forms on the page) or greater. In JavaScript, arrays are zero-based, meaning that the first form on the page is referenced with the number zero (0) as shown in the syntax example above.

### *Square Bracket Notation*

Objects can also be referenced using square bracket notation as shown below.

#### Syntax

```
window['document']['loginform']
```

```
// and
```

# JAVA SCRIPT

---

```
window['document']['forms[0]']
```

Dot notation and square bracket notation are completely interchangeable. Dot notation is much more common; however, as we will see later in the course, there are times when it is more convenient to use square bracket notation.

## Where Is JavaScript Code Written?

JavaScript code can be written inline (e.g, within HTML tags called event handlers), in script blocks, and in external JavaScript files. The page below shows examples of all three.

### *Code Sample: JavaScriptBasics/Demos/JavaScript.html*

```
<html>
<head>
  <title>JavaScript Page</title>
  <script type="text/javascript">
    window.alert("The page is loading");
  </script>
</head>
<body>
<p align="center">
  <span onclick="document.bgColor = 'red';">Red</span> |
  <span onclick="document.bgColor = 'white';">White</span>
</p>
<script type="text/javascript" src="Script.js"></script>
</body>
</html>
```

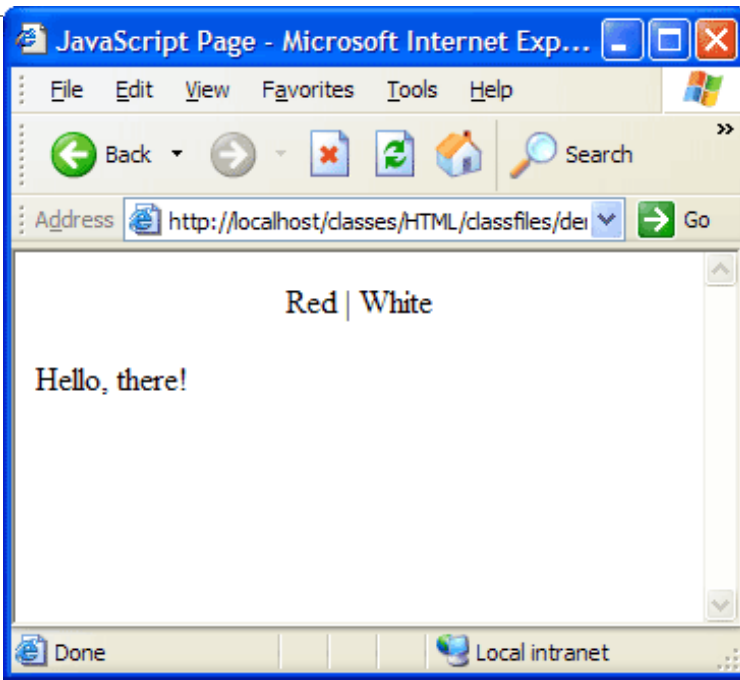
### *Code Sample: JavaScriptBasics/Demos/Script.js*

```
document.write("Hello, there!");
```

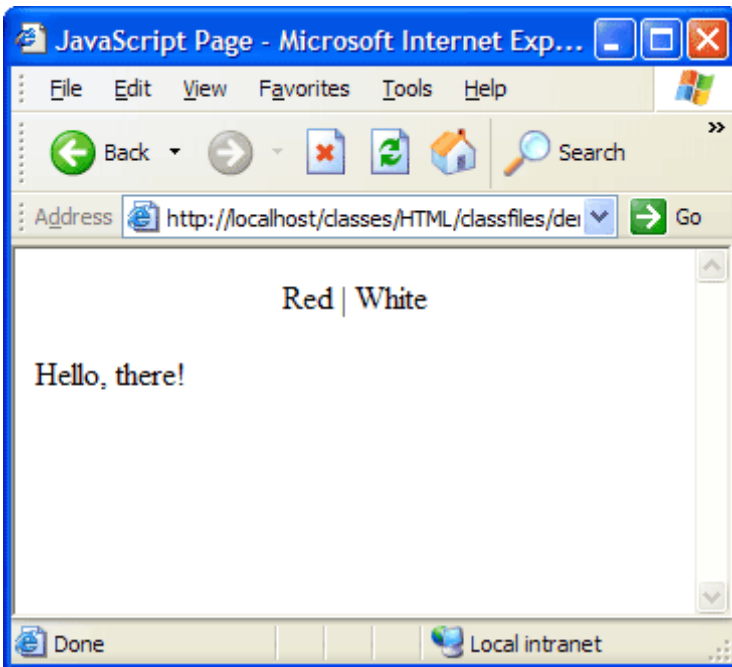
Code Explanation

As this page loads, an alert will pop up that says "The page is loading" as shown below.

## JAVA SCRIPT



After the user clicks the OK button, the page will finish loading and will appear as follows.



The text "Hello, there!" is written dynamically by the code in [JavaScriptBasics/Demos/Script.js](#). We will look at the code in this file and in [JavaScriptBasics/Demos/JavaScript.html](#) again shortly.

# JAVA SCRIPT

---

## JavaScript Objects, Methods and Properties

JavaScript is used to manipulate or get information about objects in the HTML DOM. Objects in an HTML page have methods (actions, such as opening a new window or submitting a form) and properties (attributes or qualities, such as color and size).

To illustrate objects, methods and properties, we will look at the code in [JavaScriptBasics/Demos/JavaScript2.html](#), a slightly modified version of [JavaScriptBasics/Demos/JavaScript.html](#), which we looked at earlier, and at the code in [JavaScriptBasics/Demos/Script2.js](#).

### *Code Sample: JavaScriptBasics/Demos/JavaScript2.html*

```
<html>
<head>
  <title>JavaScript Page</title>
  <script type="text/javascript">
    //Pop up an alert
    window.alert("The page is loading");
  </script>
</head>
<body>
<p align="center">
  <span onclick="document.bgColor = 'red';">Red</span> |
  <span onclick="document.bgColor = 'white';">White</span> |
  <span onclick="document.bgColor = 'green';">Green</span> |
  <span onclick="document.bgColor = 'blue';">Blue</span>
</p>
<script type="text/javascript" src="Script2.js"></script>
</body>
</html>
```

### *Code Sample: JavaScriptBasics/Demos/Script2.js*

```
/*
This script simply outputs
"Hello, there!"
to the browser.
*/
document.write("Hello, there!");
```

## **Methods**

Methods are the verbs of JavaScript. They cause things to happen.

### **window.alert()**

HTML pages are read and processed from top to bottom. The JavaScript code in the initial script block at the top of [JavaScriptBasics/Demos/JavaScript2.html](#) calls the alert() method of the window object. When the browser reads that line of code, it will pop up an alert box and will not continue processing the page until the user presses the OK button. Once the user presses the button, the alert box disappears and the rest of the page loads.

# JAVA SCRIPT

---

## **document.write()**

The write() method of the document object is used to write out code to the page as it loads. In [JavaScriptBasics/Demos/Script2.js](#), it simply writes out "Hello, there!"; however, it is more often used to write out dynamic data, such as the date and time on the user's machine.

## **Arguments**

Methods can take zero or more arguments separated by commas.

Syntax

```
object.method(argument1, argument2);
```

The alert() and write() methods shown in the example above each take only one argument: the message to show.

## **Properties**

Properties are the adjectives of JavaScript. They describe qualities of objects and, in some cases are writable (can be changed dynamically).

## **document.bgColor**

The bgColor property of the document object is read-write. Looking back at [JavaScriptBasics/Demos/JavaScript2.html](#), the four span elements use the onclick event handler to catch click events. When the user clicks on a span, JavaScript is used to change the value of the bgColor property to a new color.

## ***The Implicit window Object***

The window object is always the implicit top-level object and therefore does not have to be included in references to objects. For example, window.document.write() can be shortened to document.write(). Likewise, window.alert() can be shortened to just alert().

## ***The getElementById() Method***

A very common way to reference HTML elements is by their ID using the getElementById() method of the document object as shown in the example below.

## **Event Handlers**

In [JavaScriptBasics/Demos/JavaScript2.html](#), we used the onclick event handler to call JavaScript code that changed the background color of the page. Event handlers are attributes that force an element to "listen" for a specific event to occur. Event handlers all begin with the letters "on". The table below lists the HTML event handlers with descriptions.

## JAVA SCRIPT

---

<b>Event Handler</b>	<b>Elements Supported</b>	<b>Description</b>
onblur	a, area, button, input, label, select, textarea	the element lost the focus
onchange	input, select, textarea	the element value was changed
onclick	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer button was clicked
ondblclick	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer button was double clicked
onfocus	a, area, button, input, label, select, textarea	the element received the focus
onkeydown	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a key was pressed down
onkeypress	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a key was pressed and released
onkeyup	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a key was released
onload	frameset	all the frames have been loaded
onload	body	the document has been loaded
onmousedown	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer button was pressed down
onmousemove	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer was moved within
onmouseout	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer was moved away
onmouseover	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer was moved onto
onmouseup	All elements except applet, base, basefont, bdo, br, font, frame, frameset, head, html, iframe, isindex, meta, param, script, style, title	a pointer button was released
onreset	form	the form was reset
onselect	input, textarea	some text was selected
onsubmit	form	the form was submitted
onunload	frameset	all the frames have been removed

---

# JAVA SCRIPT

## HTML Event Handlers

Event Handler	Elements Supported	Description
onunload	body	the document has been removed

### ***Exercise: Using Event Handlers***

Duration: 15 to 25 minutes.

In this exercise, you will use some of the event handlers from the table above to allow the user to change the background color of the page.

1. Open [JavaScriptBasics/Exercises/JavaScript.html](#) for editing.
2. Modify the page so that...
  - when it is finished loading an alert pops up reading "The page has loaded!"
  - when the "Red" button is *clicked*, the background color turns red and an alert pops up reading "The background color is now Red."
  - when the "Green" button is *double-clicked*, the background color turns green and an alert pops up reading "The background color is now Green."
  - when the "Orange" button is *clicked down*, the background color turns orange and an alert pops up reading "The background color is now Orange."
  - when the mouse button is *released* over the "Blue" button, the background color turns blue and an alert pops up reading "The background color is now Blue."

### ***Code Sample: JavaScriptBasics/Exercises/JavaScript.html***

```
<html>
<head>
  <title>JavaScript Page</title>
  <script type="text/javascript">
    window.alert("The page is loading.");
  </script>
</head>
<body>
<form>
  Click the button to turn the page
  <input type="button" value="Red"/>
  <br/><br/>
  Double click the button to turn the page
  <input type="button" value="Green"/>
  <br/><br/>
  Click down on the button to turn the page
  <input type="button" value="Orange"/>
  <br/><br/>
  Release the mouse while on the button to turn the page
  <input type="button" value="Blue"/>
</form>
<hr/>
<script type="text/javascript" src="Script.js"></script>
</body>
</html>
```

1. Add functionality so that when the user presses any key, the background color turns white.



## JAVA SCRIPT

---

2. Add a "Black" button. When the user hovers over this button and presses the mouse button down, the background color should turn black. When the user releases the mouse button, the background color should turn white.

[Where is the solution?](#)

### JavaScript Basics Conclusion

In this lesson of the JavaScript tutorial, you have learned the basics of JavaScript. Now you're ready for more.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## Variables, Arrays and Operators

In this lesson of the JavaScript tutorial, you will learn...

1. To create, read and modify JavaScript variables.
2. To work with JavaScript arrays.
3. To work with JavaScript operators.

### JavaScript Variables

Variables are used to hold data in memory. JavaScript variables are declared with the `var` keyword.

```
var age;
```

Multiple variables can be declared in a single step.

```
var age, height, weight, gender;
```

After a variable is declared, it can be assigned a value.

```
age = 34;
```

Variable declaration and assignment can be done in a single step.

```
var age = 34;
```

### *A Loosely-typed Language*

JavaScript is a loosely-typed language. This means that you do not specify the data type of a variable when declaring it. It also means that a single variable can hold different data types at different times and that JavaScript can change the variable type on the fly. For example, the `age` variable above is an integer. However, the variable `strAge` below would be a string (text) because of the quotes.

```
var strAge = "34";
```

If you were to try to do a math function on `strAge` (e.g. multiply it by 4), JavaScript would dynamically change it to an integer. Although this is very convenient, it can also cause unexpected results, so be careful.

### *Storing User-Entered Data*

The following example uses the `prompt()` method of the window object to collect user input. The value entered by the user is then assigned to a variable, which is accessed when the user clicks on one of the span elements.

### *Code Sample: VariablesArraysOperators/Demos/Variables.html*

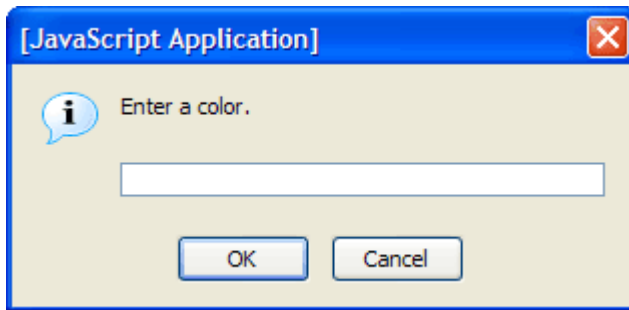
```
<html>
```

# JAVA SCRIPT

```
<head>
<title>JavaScript Variables</title>
<script type="text/javascript">
  //Pop up a prompt
  var USER_COLOR = window.prompt("Enter a color.", "");
</script>
</head>
<body>
<p align="center">
  <span onclick="document.bgColor = 'red';">Red</span> |
  <span onclick="document.bgColor = 'white';">White</span> |
  <span onclick="document.bgColor = 'green';">Green</span> |
  <span onclick="document.bgColor = 'blue';">Blue</span> |
  <span onclick="document.bgColor = USER_COLOR;">
  <script type="text/javascript">
    document.write (USER_COLOR) ;
  </script>
</span>
</p>
</body>
</html>
```

## Code Explanation

As the page loads, a prompt pops up asking the user to enter a color.



This is done with the `prompt()` method of the window object. The `prompt()` method is used to get input from the user. It takes two arguments:

1. The message in the dialog box (e.g., "Enter a color.").
2. The default value that appears in the text box. In the example above this is an empty string (e.g., "").

If the OK button is pressed, the prompt returns the value entered in the textbox. If the Cancel button or the close button (the red X) is pressed, the prompt returns null. The line below assigns whatever is returned to the variable `USER_COLOR`.

```
var USER_COLOR = window.prompt("Enter a color.", "");
```

A script block with a call to `document.write()` is then used to output the color entered by the user. This output is contained within a span element, which has an `onclick` event handler that will be used to turn the background color of the page to the user-entered color.

```
<span onclick="document.bgColor = USER_COLOR;">
```

## JAVA SCRIPT

---

```
<script type="text/javascript">
  document.write(USER_COLOR);
</script>
</span>
```

### ***Exercise: Using Variables***

Duration: 5 to 15 minutes.

In this exercise, you will practice using variables.

1. Open [VariablesArraysOperators/Exercises/Variables.html](#) for editing.
2. Below the ADD PROMPT HERE comment, write code that will prompt the user for her first name and assign the result to a variable.
3. Add a button below the Ringo button that reads "Your Name". Add functionality so that when this button is pressed an alert pops up showing the user's first name.
4. Test your solution in a browser.

### ***Code Sample: VariablesArraysOperators/Exercises/Variables.html***

```
<html>
<head>
  <title>JavaScript Variables</title>
  <script type="text/javascript">
    //ADD PROMPT HERE
  </script>
</head>
<body>
<form>
  <input type="button" value="Paul"
    onclick="alert('Paul');"/>
  <br/><br/>
  <input type="button" value="John"
    onclick="alert('John');"/>
  <br/><br/>
  <input type="button" value="George"
    onclick="alert('George');"/>
  <br/><br/>
  <input type="button" value="Ringo"
    onclick="alert('Ringo');"/>
  <br/><br/>
  <!--ADD BUTTON HERE-->
</form>
</body>
</html>
```

[Where is the solution?](#)

## **Arrays**

An array is a grouping of objects that can be accessed through subscripts. At its simplest, an array can be thought of as a list. In JavaScript, the first element of an array is considered to be at position zero (0), the second element at position one (1), and so on. Arrays are useful for storing data of similar types.

Arrays are declared using the new keyword.

## JAVA SCRIPT

---

```
var myarray = new Array();
```

It is also possible and very common to use the [] literal to declare a new Array object.

```
var myarray = [];
```

Values are assigned to arrays as follows.

```
myarray[0] = value1;
myarray[1] = value2;
myarray[2] = value3;
```

Arrays can be declared with initial values.

```
var myarray = new Array(value1, value2, value3);
//or, using the [] notation:
var myarray = [value1, value2, value3];
```

The following example is similar to the previous one, except that it prompts the user for four different colors and places each into the USER\_COLORS array. It then displays the values in the USER\_COLORS array in the spans and assigns them to document.bgColor when the user clicks on the spans.

Unlike in some languages, values in JavaScript arrays do not all have to be of the same data type.

### ***Code Sample: VariablesArraysOperators/Demos/Arrays.html***

```
<html>
<head>
<title>JavaScript Arrays</title>
<script type="text/javascript">
  //Pop up four prompts and create an array
  var USER_COLORS = new Array();
  USER_COLORS[0] = window.prompt("Choose a color.", "");
  USER_COLORS[1] = window.prompt("Choose a color.", "");
  USER_COLORS[2] = window.prompt("Choose a color.", "");
  USER_COLORS[3] = window.prompt("Choose a color.", "");
</script>
</head>
<body>
<p align="center">
  <span onclick="document.bgColor = USER_COLORS[0];">
  <script type="text/javascript">
    document.write(USER_COLORS[0]);
  </script>
  </span> |
  <span onclick="document.bgColor = USER_COLORS[1];">
  <script type="text/javascript">
    document.write(USER_COLORS[1]);
  </script>
  </span> |
  <span onclick="document.bgColor = USER_COLORS[2];">
  <script type="text/javascript">
    document.write(USER_COLORS[2]);
  </script>
  </span> |
```

## JAVA SCRIPT

```
<span onclick="document.bgColor = USER_COLORS[3];">
<script type="text/javascript">
  document.write(USER_COLORS[3]);
</script>
</span>
</p>
</body>
</html>
```

### Code Explanation

As the page loads, an array called USER\_COLORS is declared.

```
var USER_COLORS = new Array();
```

The next four lines populate the array with user-entered values.

```
USER_COLORS[0] = window.prompt("Choose a color.", "");
USER_COLORS[1] = window.prompt("Choose a color.", "");
USER_COLORS[2] = window.prompt("Choose a color.", "");
USER_COLORS[3] = window.prompt("Choose a color.", "");
```

The body of the page contains a paragraph with four span tags, the text of which is dynamically created with values from the USER\_COLORS array.

### ***Exercise: Working with Arrays***

Duration: 15 to 25 minutes.

In this exercise, you will practice working with arrays.

1. Open [VariablesArraysOperators/Exercises/Arrays.html](#) for editing.
2. Below the comment, declare a ROCK\_STARS array and populate it with four values entered by the user.
3. Add functionality to the buttons, so that alerts pop up with values from the array when the buttons are clicked.
4. Test your solution in a browser.

### ***Code Sample: VariablesArraysOperators/Exercises/Arrays.html***

```
<html>
<head>
  <title>JavaScript Arrays</title>
  <script type="text/javascript">
    /*
     Declare a ROCK_STARS array and populate it with
     four values entered by the user.
    */
  </script>
</head>
<body>
<form>
  <input type="button" value="Favorite"/>
  <br/><br/>
  <input type="button" value="Next Favorite"/>
  <br/><br/>
  <input type="button" value="Next Favorite"/>
```

## JAVA SCRIPT

```
<br/><br/>
<input type="button" value="Next Favorite"/>
</form>
</body>
</html>
```

[Where is the solution?](#)

### ***Associative Arrays***

Whereas regular (or enumerated) arrays are indexed numerically, associative arrays are indexed using names as keys. The advantage of this is that the keys can be meaningful, which can make it easier to reference an element in an array. The example below illustrates how an associative array is used.

#### ***Code Sample: VariablesArraysOperators/Demos/AssociativeArrays.html***

```
<html>
<head>
  <title>JavaScript Arrays</title>
  <script type="text/javascript">
    var BEATLES = [];
    BEATLES["singer1"] = "Paul";
    BEATLES["singer2"] = "John";
    BEATLES["guitarist"] = "George";
    BEATLES["drummer"] = "Ringo";
  </script>
</head>
<body>
  <p align="center">
    <script type="text/javascript">
      document.write(BEATLES["singer1"]);
      document.write(BEATLES["singer2"]);
      document.write(BEATLES["guitarist"]);
      document.write(BEATLES["drummer"]);
    </script>
  </p>
</body>
</html>
```

### ***Array Properties and Methods***

The tables below show some of the most useful array properties and methods. All of the examples assume an array called BEATLES that holds "Paul", "John", "George", and "Ringo".

```
var BEATLES = ["Paul", "John", "George", "Ringo"];
```

#### Array Properties

Property	Description	Example
length	Holds the number of elements in an array.	BEATLES.length // 4

#### Array Methods

Property	Description	Example
join(delimiter)	Returns a delimited list of the items indexed with integers in the array. The default delimiter is a comma.	BEATLES.join(":") // Paul:John:George:Ringo

# JAVA SCRIPT

## Array Methods

Property	Description	Example
pop()	Removes the last item in an array and returns its value.	BEATLES.pop() // Returns Ringo
shift()	Removes the first item in an array and returns its value.	BEATLES.shift() // Returns Paul
slice(start, end)	Returns a subarray from start to end. If end is left out, it includes the remainder of the array.	BEATLES.slice(1, 2) //Returns [John, George]
splice(start, count)	Removes count items from start in the array and returns the resulting array.	BEATLES.splice(1, 2) //Returns [Paul, Ringo]

## JavaScript Operators

### Arithmetic Operators

Operator	Description
----------	-------------

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
++	Increment by one
--	Decrement by one

### Assignment Operators

Operator	Description
----------	-------------

=	Assignment
+=	One step addition and assignment (a+=3 is the same as a=a+3)
-=	One step subtraction and assignment (a-=3 is the same as a=a-3)
*=	One step multiplication and assignment (a*=3 is the same as a=a*3)
/=	One step division and assignment (a/=3 is the same as a=a/3)
%=	One step modulus and assignment (a%=3 is the same as a=a%3)

### String Operators

Operator	Description
----------	-------------

+	Concatenation (var greeting = "Hello " + firstname;)
+=	One step concatenation and assignment (var greeting = "Hello "; greeting += firstname;)

### Ternary Operator

Operator	Description
----------	-------------

?:	Conditional evaluation (var evenOrOdd = (number % 2 == 0) ? "even" : "odd";)
----	--

The following code sample shows these operators in use.

### ***Code Sample: VariablesArraysOperators/Demos/Operators.html***

```
<html>
<head>
  <title>JavaScript Operators</title>
  <script type="text/javascript">
```



## JAVA SCRIPT

```
var USER_NUM1 = window.prompt("Choose a number.", "");
alert("You chose " + USER_NUM1);
var USER_NUM2 = window.prompt("Choose another number.", "");
alert("You chose " + USER_NUM2);
var NUMS_ADDED = USER_NUM1 + Number(USER_NUM2);
var NUMS_SUBTRACTED = USER_NUM1 - USER_NUM2;
var NUMS_MULTIPLIED = USER_NUM1 * USER_NUM2;
var NUMS_DIVIDED = USER_NUM1 / USER_NUM2;
var NUMS_MODULUSED = USER_NUM1 % USER_NUM2;
</script>
</head>
<body>
<p style="text-align:center; font-size:large">
<script type="text/javascript">
document.write(USER_NUM1 + " + " + USER_NUM2 + " = ");
document.write(NUMS_ADDED + "<br/>");
document.write(USER_NUM1 + " - " + USER_NUM2 + " = ");
document.write(NUMS_SUBTRACTED + "<br/>");
document.write(USER_NUM1 + " * " + USER_NUM2 + " = ");
document.write(NUMS_MULTIPLIED + "<br/>");
document.write(USER_NUM1 + " / " + USER_NUM2 + " = ");
document.write(NUMS_DIVIDED + "<br/>");
document.write(USER_NUM1 + " % " + USER_NUM2 + " = ");
document.write(NUMS_MODULUSED + "<br/>");
</script>
</p>
</body>
</html>
```

### Code Explanation

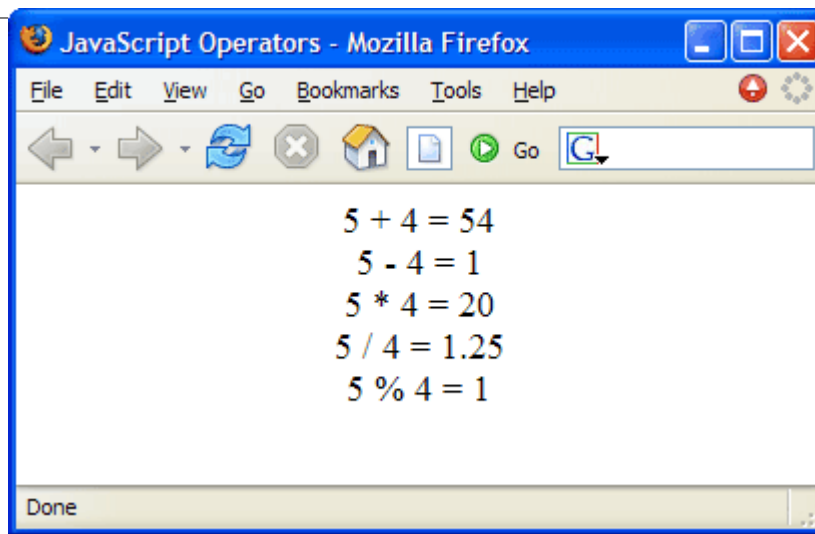
The file above illustrates the use of the concatenation operator and several math operators. It also illustrates a potential problem with loosely-typed languages. This page is processed as follows:

1. The user is prompted for a number and the result is assigned to USER\_NUM1.
2. An alert pops up telling the user what number she entered. The concatenation operator (+) is used to combine two strings: "You chose " and the number entered by the user. Note that all user-entered data is always treated as a string of text, even if the text consists of only digits.
3. The user is prompted for another number and the result is assigned to USER\_NUM2.
4. Another alert pops up telling the user what number she entered.
5. Five variables are declared and assigned values.

```
var NUMS_ADDED = USER_NUM1 + USER_NUM2;
var NUMS_SUBTRACTED = USER_NUM1 - USER_NUM2;
var NUMS_MULTIPLIED = USER_NUM1 * USER_NUM2;
var NUMS_DIVIDED = USER_NUM1 / USER_NUM2;
var NUMS_MODULUSED = USER_NUM1 % USER_NUM2;
```

6. The values the variables contain are output to the browser.

# JAVA SCRIPT



So,  $5 + 4$  is 54! It is when 5 and 4 are strings, and, as stated earlier, all user-entered data is treated as a string. In the lesson on JavaScript Functions, you will learn how to convert a string to a number.

## ***Exercise: Working with Operators***

Duration: 15 to 25 minutes.

In this exercise, you will practice working with JavaScript operators.

1. Open [VariablesArraysOperators/Exercises/Operators.html](#) for editing.
2. Add code to prompt the user for the number of CDs she owns of her favorite and second favorite rockstars'.
3. In the body, let the user know how many more of her favorite rockstar's CDs she has than of her second favorite rockstar's CDs.
4. Test your solution in a browser.

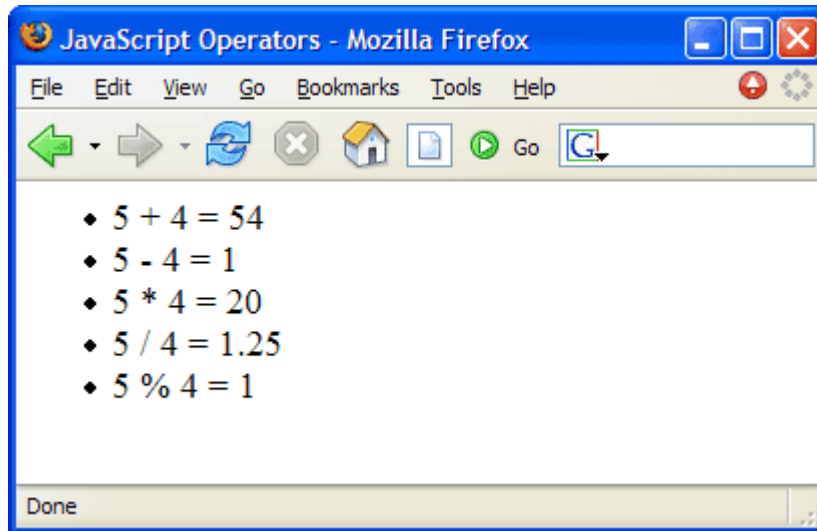
## ***Code Sample: VariablesArraysOperators/Exercises/Operators.html***

```
<html>
<head>
  <title>JavaScript Operators</title>
  <script type="text/javascript">
    var ROCK_STARS = [];
    ROCK_STARS[0] = prompt("Who is your favorite rock star?", "");
    /*
    Ask the user how many of this rockstar's CDs she owns and store
    the result in a variable.
    */
    ROCK_STARS[1] = prompt("And your next favorite rock star?", "");
    /*
    Ask the user how many of this rockstar's CDs she owns and store
    the result in a variable.
    */
  </script>
</head>
<body>
<!--
  Let the user know how many more of her favorite rockstar's CDs
```

## JAVA SCRIPT

```
she has than of her second favorite rockstar's CDs.  
-->  
</body>  
</html>
```

1. Open [VariablesArraysOperators/Exercises/Operators-challenge.html](#) for editing.
2. Modify it so that it outputs an unordered list as shown below.



[Where is the solution?](#)

## Variables, Arrays and Operators Conclusion

In this lesson of the JavaScript tutorial, you have learned to work with JavaScript variables, arrays and operators.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## JavaScript Functions

In this lesson of the JavaScript tutorial, you will learn...

1. To work with some of JavaScript's built-in functions.
2. To create your own functions.
3. To return values from functions.

### Built-in Functions

JavaScript has a number of built-in functions. We will examine some of them in this section.

#### *Number(object)*

The Number() function takes one argument: an object, which it attempts to convert to a number. If it cannot, it returns NaN, for "Not a Number."

#### *Code Sample: JavaScriptFunctions/Demos/Number.html*

```
<html>
<head>
  <title>Number() Function</title>
  <script type="text/javascript">
    var STR_NUM1 = "1";
    var STR_NUM2 = "2";
    var STR_SUM = STR_NUM1 + STR_NUM2; //returns 12
    alert(STR_SUM);

    var INT_NUM1 = Number(STR_NUM1);
    var INT_NUM2 = Number(STR_NUM2);
    var INT_SUM = INT_NUM1 + INT_NUM2; //returns 3
    alert(INT_SUM);
  </script>
</head>
<body>
  Nothing to show here.
</body>
</html>
```

#### Code Explanation

Because STR\_NUM1 and STR\_NUM2 are both strings, the + operator concatenates them, resulting in "12".

```
var STR_NUM1 = "1";
var STR_NUM2 = "2";
var STR_SUM = STR_NUM1 + STR_NUM2; //returns 12
alert(STR_SUM);
```

After the Number() function has been used to convert the strings to numbers, the + operator performs addition, resulting in 3.

```
var INT_NUM1 = Number(STR_NUM1);
var INT_NUM2 = Number(STR_NUM2);
var INT_SUM = INT_NUM1 + INT_NUM2; //returns 3
```

```
alert(INT_SUM);
```

## ***String(object)***

The String() function takes one argument: an object, which it converts to a string.

### ***Code Sample: JavaScriptFunctions/Demos/String.html***

```
<html>
<head>
  <title>String() Function</title>
  <script type="text/javascript">
    var INT_NUM1 = 1;
    var INT_NUM2 = 2;
    var INT_SUM = INT_NUM1 + INT_NUM2; //returns 3
    alert(INT_SUM);

    var STR_NUM1 = String(INT_NUM1);
    var STR_NUM2 = String(INT_NUM2);
    var STR_SUM = STR_NUM1 + STR_NUM2; //returns 12
    alert(STR_SUM);
  </script>
</head>
<body>
  Nothing to show here.
</body>
</html>
```

#### **Code Explanation**

Because INT\_NUM1 and INT\_NUM2 are both numbers, the + operator performs addition, resulting in 3.

```
var INT_NUM1 = 1;
var INT_NUM2 = 2;
var INT_SUM = INT_NUM1 + INT_NUM2; //returns 3
alert(INT_SUM);
```

After the String() function has been used to convert the numbers to string, the + operator performs concatenation, resulting in "12".

```
var STR_NUM1 = String(INT_NUM1);
var STR_NUM2 = String(INT_NUM2);
var STR_SUM = STR_NUM1 + STR_NUM2; //returns 12
alert(STR_SUM);
```

## ***isNaN(object)***

The isNaN() function takes one argument: an object. The function checks if the object is *not* a number (or cannot be converted to a number). It returns true if the object is not a number and false if it is a number.

### ***Code Sample: JavaScriptFunctions/Demos/isNaN.html***

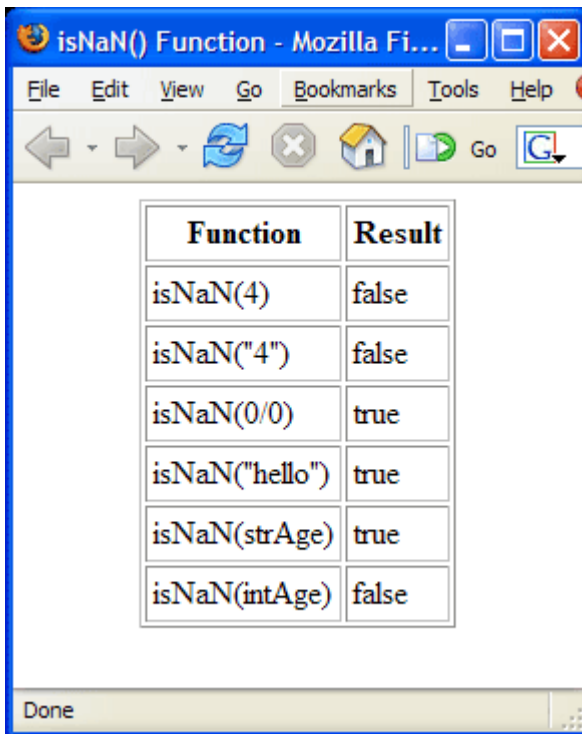
```
<html>
<head>
```

## JAVA SCRIPT

```
<title>isNaN() Function</title>
</head>
<body>
<table border="1" cellpadding="3" align="center">
<tr>
<th>Function</th><th>Result</th>
</tr>
<script type="text/javascript">
document.write("<tr><td>isNaN(4)</td>");
document.write("<td>" + isNaN(4) + "</td></tr>");
document.write("<tr><td>isNaN(\"4\")</td>");
document.write("<td>" + isNaN("4") + "</td></tr>");
document.write("<tr><td>isNaN(0/0)</td>");
document.write("<td>" + isNaN(0/0) + "</td></tr>");
document.write("<tr><td>isNaN(\"hello\")</td>");
document.write("<td>" + isNaN("hello") + "</td></tr>");
var AGE_STR = "twelve";
document.write("<tr><td>isNaN(AGE_STR)</td>");
document.write("<td>" + isNaN(AGE_STR) + "</td></tr>");
var AGE_INT = 12;
document.write("<tr><td>isNaN(AGE_INT)</td>");
document.write("<td>" + isNaN(AGE_INT) + "</td></tr>");
</script>
</table>
</body>
</html>
```

Code Explanation

The output will look like this:



Function	Result
isNaN(4)	false
isNaN("4")	false
isNaN(0/0)	true
isNaN("hello")	true
isNaN(strAge)	true
isNaN(intAge)	false

### *parseFloat() and parseInt()*

The `parseFloat()` function takes one argument: a string. If the string begins with a number, the function reads through the string until it finds the end of the number, hacks off the remainder of the string, and returns the result. If the string does not begin with a number, the function returns NaN.

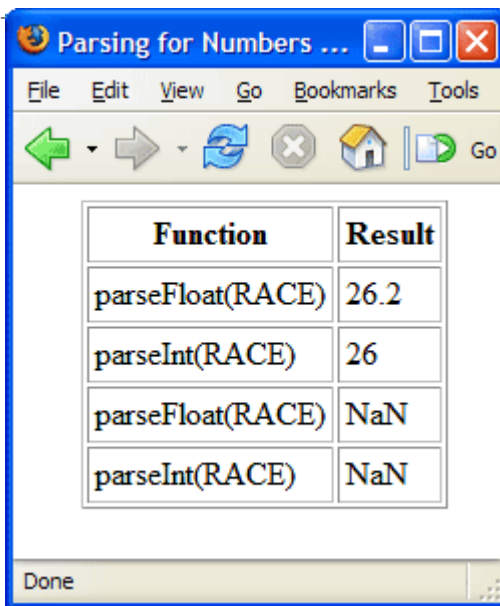
The `parseInt()` function also takes one argument: a string. If the string begins with an integer, the function reads through the string until it finds the end of the integer, hacks off the remainder of the string, and returns the result. If the string does not begin with an integer, the function returns NaN.

### *Code Sample: JavaScriptFunctions/Demos/ParsingNumbers.html*

```
<html>
<head>
  <title>Parsing for Numbers</title>
</head>
<body>
<table border="1" cellpadding="3" align="center">
<tr>
  <th>Function</th><th>Result</th>
</tr>
<script type="text/javascript">
  var RACE = "26.2 miles";
  document.write("<tr><td>parseFloat (RACE)</td>");
  document.write("<td>" + parseFloat (RACE) + "</td></tr>");
  document.write("<tr><td>parseInt (RACE)</td>");
  document.write("<td>" + parseInt (RACE) + "</td></tr>");
  RACE = "Marathon";
  document.write("<tr><td>parseFloat (RACE)</td>");
  document.write("<td>" + parseFloat (RACE) + "</td></tr>");
  document.write("<tr><td>parseInt (RACE)</td>");
  document.write("<td>" + parseInt (RACE) + "</td></tr>");
</script>
</table>
</body>
</html>
```

#### Code Explanation

The output will look like this:



Function	Result
parseFloat(RACE)	26.2
parseInt(RACE)	26
parseFloat(RACE)	NaN
parseInt(RACE)	NaN

## Built-in Functions vs. Methods

Methods and functions are similar in that they both make things happen. They are also syntactically similar. The major difference is that methods are tied to an object; whereas, functions are not. For example, `alert()` is a method of the window object; whereas `parseInt()` is a standalone function.

### *Exercise: Working with Built-in Functions*

Duration: 15 to 25 minutes.

In this exercise, you will practice working with JavaScript's built-in functions.

1. Open [JavaScriptFunctions/Exercises/BuiltinFunctions.html](#) for editing.
2. Modify the file so that it outputs the sum of the two numbers entered by the user.

### *Code Sample: JavaScriptFunctions/Exercises/BuiltinFunctions.html*

```
<html>
<head>
  <title>JavaScript Operators</title>
  <script type="text/javascript">
    var USER_NUM1, USER_NUM2, NUMS_ADDED;
    USER_NUM1 = window.prompt("Choose a number.", "");
    alert("You chose " + USER_NUM1);
    USER_NUM2 = window.prompt("Choose another number.", "");
    alert("You chose " + USER_NUM2);
    NUMS_ADDED = USER_NUM1 + USER_NUM2;
  </script>
</head>
<body>
<p style="text-align:center; font-size:large">
  <script type="text/javascript">
    document.write(USER_NUM1 + " + " + USER_NUM2 + " = ");
    document.write(NUMS_ADDED + "<br/>");
  </script>
```

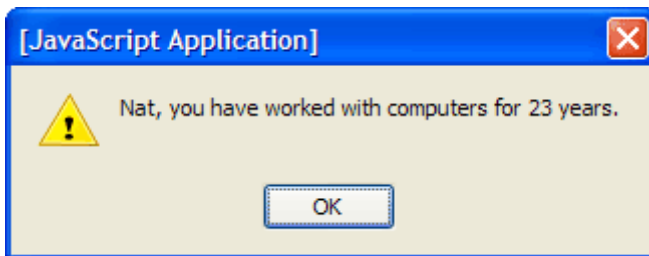
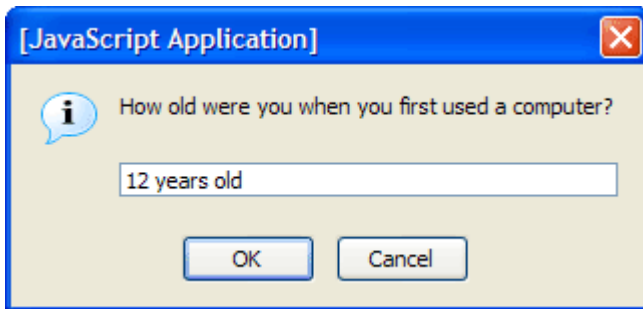
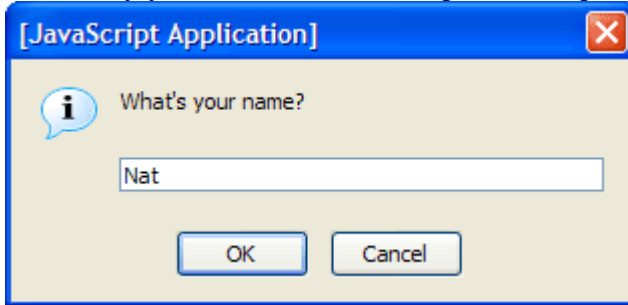


## JAVA SCRIPT

```
</p>  
</body>  
</html>
```

Create a new HTML file that prompts the user for his name, the age at which he first worked on a computer, and his current age. After gathering this information, pop up an alert that tells the user

how many years he's been working on a computer. The images below show the steps:



Notice that the program is able to deal with numbers followed by strings (e.g. "12 years old").

[Where is the solution?](#)

## User-defined Functions

Writing functions makes it possible to reuse code for common tasks. Functions can also be used to hide complex code. For example, an experienced developer can write a function for performing a complicated task. Other developers do not need to know how that function works; they only need to know how to call it.

# JAVA SCRIPT

## ***Function Syntax***

JavaScript functions generally appear in the head of the page or in external JavaScript files. A function is written using the function keyword followed by the name of the function.

Syntax

```
function doSomething() {  
    //function statements go here  
}
```

As you can see, the body of the function is contained within curly brackets ({}). The following example demonstrates the use of simple functions.

### ***Code Sample: JavaScriptFunctions/Demos/SimpleFunctions.html***

```
<html>  
<head>  
  <title>JavaScript Simple Functions</title>  
  <script type="text/javascript">  
    function changeBgRed() {  
      document.bgColor = "red";  
    }  
  
    function changeBgWhite() {  
      document.bgColor = "white";  
    }  
  </script>  
</head>  
<body>  
<p align="center">  
  <span onclick="changeBgRed();">Red</span> |  
  <span onclick="changeBgWhite();">White</span>  
</p>  
</body>  
</html>
```

## ***Passing Values to Functions***

The functions above aren't very useful because they always do the same thing. Every time we wanted to add another color, we would have to write another function. Also, if we want to modify the behavior, we will have to do it in each function. The example below shows how to create a single function to handle changing the background color.

### ***Code Sample: JavaScriptFunctions/Demos/PassingValues.html***

```
<html>  
<head>  
  <title>JavaScript Simple Functions</title>  
  <script type="text/javascript">  
    function changeBg(color) {  
      document.bgColor = color;  
    }  
  </script>  
</head>  
<body>  
<p align="center">
```

## JAVA SCRIPT

```
<span onclick="changeBg('red');">Red</span> |  
<span onclick="changeBg('white');">White</span>  
</p>  
</body>  
</html>
```

### Code Explanation

As you can see, when calling the `changeBG()` function, we pass a value (e.g, 'red'), which is assigned to the color variable. We can then refer to the color variable throughout the function. Variables created in this way are called function arguments or parameters. A function can have any number of arguments, separated by commas.

### *A Note on Variable Scope*

Variables created through function arguments or declared within a function with `var` are local to the function, meaning that they cannot be accessed outside of the function.

Variables declared with `var` outside of a function and variables that are used without being declared are global, meaning that they can be used anywhere on the page.

### *Exercise: Writing a JavaScript Function*

Duration: 15 to 25 minutes.

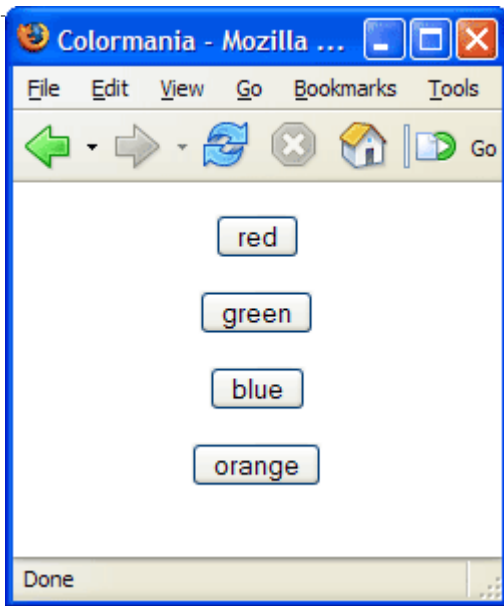
In this exercise, you will modify a page called [ColorMania.html](#), which will contain a form with four buttons. Each button will show the name of a color (e.g, red) and, when clicked, call a function that changes the background color. The buttons you will create will be of type `button`. For example,

```
<input type="button" value="red" onclick="functionCall();">
```

1. Open [JavaScriptFunctions/Exercises/ColorMania.html](#) for editing.
2. Write code to prompt the user for her name.
3. Write a function called `changeBg()` that changes the background color and then pops up an alert telling the user, by name, what the new background color is.
4. In the form, add four buttons that, when clicked, call the `changeBg()` function and pass it a color value.

The resulting page should look like this:

## JAVA SCRIPT



### ***Code Sample: JavaScriptFunctions/Exercises/ColorMania.html***

```
<html>
<head>
  <title>Colormania</title>
  <script type="text/javascript">
    //PROMPT USER FOR NAME

    /*
    Write a function called changeBg() that changes the background
    color and then pops up an alert telling the user, by name, what
    the new background color is.
    */
  </script>
</head>
<body>
<form style="text-align:center">
  <!--ADD BUTTONS HERE-->
</form>
</body>
</html>
```

Add another button called "custom" that, when clicked, prompts the user for a color, then changes the background color to the user-entered color and alerts the user to the change.

[Where is the solution?](#)

### ***Returning Values from Functions***

The return keyword is used to return values from functions as the following example illustrates.

### ***Code Sample: JavaScriptFunctions/Demos/ReturnValue.html***

```
<html>
<head>
<title>Returning a Value</title>
<script type="text/javascript">
```

## JAVA SCRIPT

```
function setBgColor(){
  document.bgColor = prompt("Set Background Color:", "");
}

function getBgColor(){
  return document.bgColor;
}
</script>
</head>

<body>
<form>
  <input type="button" value="Set Background Color"
    onclick="setBgColor();" >
  <input type="button" value="Get Background Color"
    onclick="alert(getBgColor());" >
</form>
</body>
</html>
```

### Code Explanation

When the user clicks on the "Get Background Color" button, an alert pops up with a value returned from the getBgColor() function. This is a very simple example. Generally, functions that return values are a bit more involved. We'll see many more functions that return values throughout the course.

## JavaScript Functions Conclusion

In this lesson of the JavaScript tutorial, you have learned to work with JavaScript's built-in functions and to create functions of your own.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

# JAVA SCRIPT

## Built-In JavaScript Objects

---

In this lesson of the JavaScript tutorial, you will learn...

1. To work with the built-in String object.
2. To work with the built-in Math object.
3. To work with the built-in Date object.

JavaScript has some predefined, built-in objects that do not fit into the [HTML DOM](#), meaning that they are not direct descendants of the window object.

## String

In JavaScript, there are two types of string data types: primitive strings and *String* objects. String objects have many methods for manipulating and parsing strings of text. Because these methods are available to primitive strings as well, in practice, there is no need to differentiate between the two types of strings.

Some common string properties and methods are shown below. In all the examples, the variable MY\_STRING contains "Webucator".

Common String Properties		
Property	Description	Example
length	Read-only value containing the number of characters in the string.	<code>MY_STRING.length</code> //Returns 9

Common String Methods		
Method	Description	Example
<code>charAt(position)</code>	Returns the character at the specified position.	<code>MY_STRING.charAt(4)</code> //Returns c
<code>charCodeAt(position)</code>	Returns the Unicode character code of the character at the specified position.	<code>MY_STRING.charCodeAt(4)</code> //Returns 99
<code>fromCharCode(characterCodes)</code>	Returns the text representation of the specifies comma-delimited character codes. Used with String rather than a specific String object.	<code>String.fromCharCode(169)</code> //Returns Â©
<code>indexOf(substring, startPosition)</code>	Searches from startPosition for substring. Returns the position at which the substring is found. If substring is not found, returns -1.	<code>MY_STRING.indexOf("cat");</code> //Returns 4  <code>MY_STRING.indexOf("cat", 5);</code> //Returns -1
<code>lastIndexOf(substring, endPosition)</code>	Searches from the end of the string for substring until endPosition is reached. Returns the	<code>MY_STRING.lastIndexOf("cat");</code> //Returns 4  <code>MY_STRING.lastIndexOf("cat", 5);</code>

---

# JAVA SCRIPT

---

## Common String Methods

Method	Description	Example
	position at which the substring is found. If substring is not found, returns -1.	//Returns 4
<code>substring(startPosition, endPosition)</code>	Returns the substring beginning at <code>startPosition</code> and ending with the character before <code>endPosition</code> . <code>endPosition</code> is optional. If it is excluded, the substring continues to the end of the string.	<pre>MY_STRING.substring(4, 7); //Returns cat MY_STRING.substring(4); //Returns cator</pre>
<code>substr(startPosition, length)</code>	Returns the substring of <code>length</code> characters beginning at <code>startPosition</code> . <code>length</code> is optional. If it is excluded, the substring continues to the end of the string.	<pre>MY_STRING.substr(4, 3); //Returns cat MY_STRING.substr(4); //Returns cator</pre>
<code>slice(startPosition, endPosition)</code>	Same as <code>substring(startPosition, endPosition)</code> .	<pre>MY_STRING.slice(4, 7); //Returns cat</pre>
<code>slice(startPosition, positionFromEnd)</code>	<code>positionFromEnd</code> is a negative integer. Returns the the substring beginning at <code>startPosition</code> and ending <code>positionFromEnd</code> characters from the end of the string.	<pre>MY_STRING.slice(4, -2); //Returns cat</pre>
<code>split(delimiter)</code>	Returns an array by splitting a string on the specified delimiter.	<pre>var s = "A,B,C,D"; var a = s.split(","); document.write(a[2]); //Returns C</pre>
<code>toLowerCase()</code>	Returns the string in all lowercase letters.	<pre>MY_STRING.toLowerCase() //Returns webucator</pre>
<code>toUpperCase()</code>	Returns the string in all uppercase letters.	<pre>MY_STRING.toUpperCase(); //Returns WEBUCATOR</pre>

You can see these examples in a browser by opening [BuiltInObjects/Demos/StringPropertiesAndMethods.html](#).

## Math

The *Math* object is a built-in static object. The *Math* object's properties and methods are accessed directly (e.g, `Math.PI`) and are used for performing complex math operations. Some common math properties and methods are shown below.

# JAVA SCRIPT

## Common Math Properties

Property	Description	Example
Math.PI	Pi ( )	Math.PI; //3.141592653589793
Math.SQRT2	Square root of 2.	Math.SQRT2; //1.4142135623730951

## Common Math Methods

Method	Description	Example
Math.abs (number)	Absolute value of number.	Math.abs (-12); //Returns 12
Math.ceil (number)	number rounded up.	Math.ceil (5.4); //Returns 6
Math.floor (number)	number rounded down.	Math.floor (5.6); //Returns 5
Math.max (numbers)	Highest Number in numbers.	Math.max (2, 5, 9, 3); //Returns 9
Math.min (numbers)	Lowest Number in numbers.	Math.min (2, 5, 9, 3); //Returns 2
Math.pow (number, power)	number to the power of power.	Math.pow (2, 5); //Returns 32
Math.round (number)	Rounded number.	Math.round (2.5); //Returns 3
Math.random ()	Random number between 0 and 1.	Math.random (); //Returns random //number from 0 to 1

You can see these examples in a browser by opening [BuiltInObjects/Demos/MathPropertiesAndMethods.html](#).

## Method for Generating Random Integers

```
var LOW = 1;  
var HIGH = 10;  
var RND1 = Math.random ();  
var RND2 = Math.round (RND1 * (HIGH - LOW) + 1);
```

## Date

The *Date* object has methods for manipulating dates and times. JavaScript stores dates as the number of milliseconds since January 1, 1970. The sample below shows the different methods of creating date objects, all of which involve passing arguments to the Date() constructor.

### Code Sample: BuiltInObjects/Demos/DateObject.html

```
<html>  
<head>  
<title>Date Object</title>  
</head>  
<body>  
<h1>Date Object</h1>  
<h2>New Date object with current date and time</h2>  
<pre>  
//Syntax: new Date ();
```



# JAVA SCRIPT

```
var NOW = new Date();
</pre>
<b>Result:</b>
<script type="text/javascript">
  var NOW = new Date();
  document.write(NOW);
</script>

<h2>New Date object with specific date and time</h2>
<pre>
//Syntax: new Date("month dd, yyyy hh:mm:ss");
var RED_SOX_WINS = new Date("October 21, 2004 12:01:00");
</pre>
<b>Result:</b>
<script type="text/javascript">
  var RED_SOX_WINS = new Date("October 21, 2004 12:01:00");
  document.write(RED_SOX_WINS);
</script>

<pre>
//Syntax: new Date(yyyy, mm, dd, hh, mm, ss, ms);
RED_SOX_WINS = new Date(2004, 9, 21, 12, 01, 00, 00);
</pre>
<b>Result:</b>
<script type="text/javascript">
  RED_SOX_WINS = new Date(2004, 9, 21, 12, 01, 00, 00);
  document.write(RED_SOX_WINS);
</script>
</body>
</html>
```

## Code Explanation

This page is shown in a browser below.

A few things to note:

- To create a Date object containing the current date and time, the Date() constructor takes no arguments.
- When passing the date as a string to the Date() constructor, the time portion is optional. If it is not included, it defaults to 00:00:00. Also, other date formats are acceptable (e.g. "10/21/2004" and "10-04-2004").
- When passing date parts to the Date() constructor, dd, hh, mm, ss, and ms are all optional. The default of each is 0.
- Months are numbered from 0 (January) to 11 (December). In the example above, 9 represents October.

Some common date methods are shown below. In all the examples, the variable RIGHT\_NOW contains "Thu Apr 14 00:23:54:650 EDT 2005".

## Common Date Methods

Method	Description	Example
getDate()	Returns the day of the month (1-31).	RIGHT_NOW.getDate(); //Returns 14
getDay()	Returns the day of the week as a	RIGHT_NOW.getDay();

# JAVA SCRIPT

---

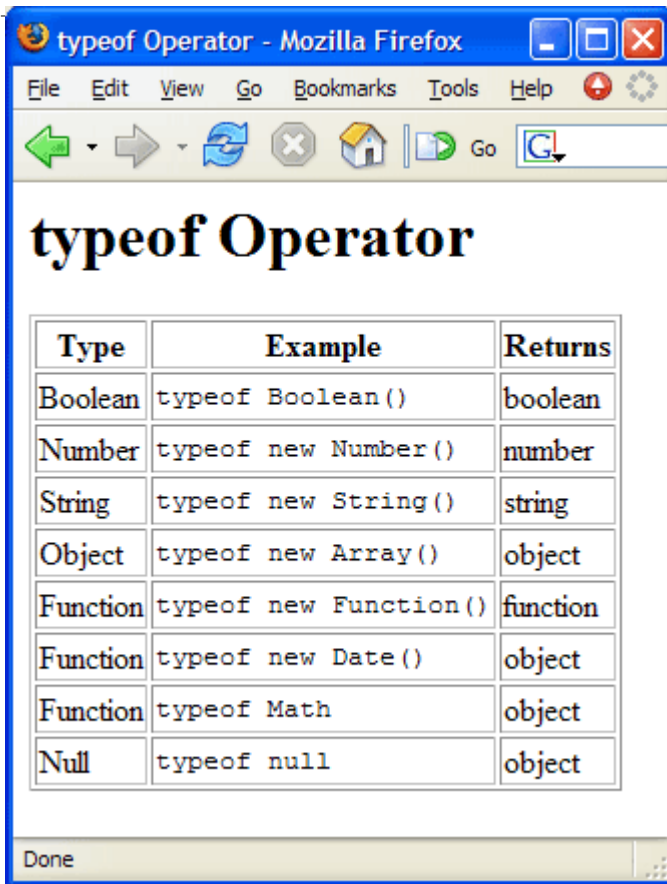
## Common Date Methods

Method	Description	Example
	number (0-6, 0=Sunday, 6=Saturday).	//Returns 4
<code>getMonth()</code>	Returns the month as a number (0-11, 0=January, 11=December).	<code>RIGHT_NOW.getMonth();</code> //Returns 3
<code>getFullYear()</code>	Returns the four-digit year.	<code>RIGHT_NOW.getFullYear();</code> //Returns 2005
<code>getHours()</code>	Returns the hour (0-23).	<code>RIGHT_NOW.getHours();</code> //Returns 0
<code>getMinutes()</code>	Returns the minute (0-59).	<code>RIGHT_NOW.getMinutes();</code> //Returns 23
<code>getSeconds()</code>	Returns the second (0-59).	<code>RIGHT_NOW.getSeconds();</code> //Returns 54
<code>getMilliseconds()</code>	Returns the millisecond (0-999).	<code>RIGHT_NOW.getMilliseconds();</code> //Returns 650
<code>getTime()</code>	Returns the number of milliseconds since midnight January 1, 1970.	<code>RIGHT_NOW.getTime();</code> //Returns 1113452634650
<code>getTimezoneOffset()</code>	Returns the time difference in minutes between the user's computer and GMT.	<code>RIGHT_NOW.getTimezoneOffset();</code> //Returns 240
<code>toLocaleString()</code>	Returns the Date object as a string.	<code>RIGHT_NOW.toLocaleString();</code> //Returns Thursday, April 14, //2005 12:23:54 AM
<code>toGMTString()</code>	Returns the Date object as a string in GMT timezone.	<code>RIGHT_NOW.toGMTString();</code> //Returns Thu, 14 Apr 2005 //04:23:54 UTC

You can see these examples in a browser by opening [BuiltInObjects/Demos/DateMethods.html](#).

## typeof Operator

The `typeof` operator is used to find out the type of a piece of data. The screenshot below shows what the `typeof` operator returns for different data types.



Some languages have functions that return the the month as a string. JavaScript doesn't have such a built-in function. The sample below shows a user-defined function that handles this and how the getMonth() method of a Date object can be used to get the month.

### ***Code Sample: BuiltInObjects/Demos/MonthAsString.html***

```
<html>
<head>
<title>Month As String</title>
<script type="text/javascript">
function monthAsString(num) {
var months = [];
months[0] = "January";
months[1] = "February";
months[2] = "March";
months[3] = "April";
months[4] = "May";
months[5] = "June";
months[6] = "July";
months[7] = "August";
months[8] = "September";
months[9] = "October";
months[10] = "November";
months[11] = "December";

return months[num-1];
}
```

## JAVA SCRIPT

```
function enterMonth(){
    var userMonth = prompt("What month were you born?", "");
    alert("You were born in " + monthAsString(userMonth) + ".");
}

function getCurrentMonth(){
    var today = new Date();
    alert(monthAsString(today.getMonth()+1));
}
</script>
</head>
<body>
<form>
    <input type="button" value="CHOOSE MONTH" onclick="enterMonth();">
    <input type="button" value="GET CURRENT MONTH" onclick="getCurrentMonth();">
</form>
</body>
</html>
```

### ***Exercise: Returning the Day of the Week as a String***

Duration: 15 to 25 minutes.

In this exercise, you will create a function that returns the day of the week as a string.

1. Open [BuiltInObjects/Exercises/DateUDFs.html](#) for editing.
2. Write a dayAsString() function that returns the day of the week as a string.
3. Write an enterDay() function that prompts the user for the day of the week and then alerts the string value of that day by calling the dayAsString() function.
4. Write a getCurrentDay() function that alerts today's actual day of the week according to the user's machine.
5. Add a "CHOOSE DAY" button that calls the enterDay() function.
6. Add a "GET CURRENT DAY" button that calls the getCurrentDay() function.
7. Test your solution in a browser.

[Where is the solution?](#)

## **Built-In JavaScript Objects Conclusion**

In this lesson of the JavaScript tutorial, you have learned to work with some of JavaScript's most useful built-in objects.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## Conditionals and Loops

In this lesson of the JavaScript tutorial, you will learn...

1. To write if - else if - else blocks.
2. To write switch / case blocks.
3. To return values from functions.
4. To work with loops in JavaScript.

## Conditionals

There are two types of conditionals in JavaScript.

1. if - else if - else
2. switch / case

### *if - else if - else Conditions*

Syntax

```
if (conditions) {  
  statements;  
} else if (conditions) {  
  statements;  
} else {  
  statements;  
}
```

Like with functions, each part of the if - else if - else block is contained within curly brackets ({}). There can be zero or more else if blocks. The else block is optional.

Comparison Operators

Operator	Description
==	Equals
!=	Doesn't equal
===	Strictly equals
!==	Doesn't strictly equal
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

Logical Operators

Operator	Description
&&	and (a == b && c != d)
	or (a == b    c != d)
!	not !(a == b    c != d)

The example below shows a function using and if - else if - else condition.

## JAVA SCRIPT

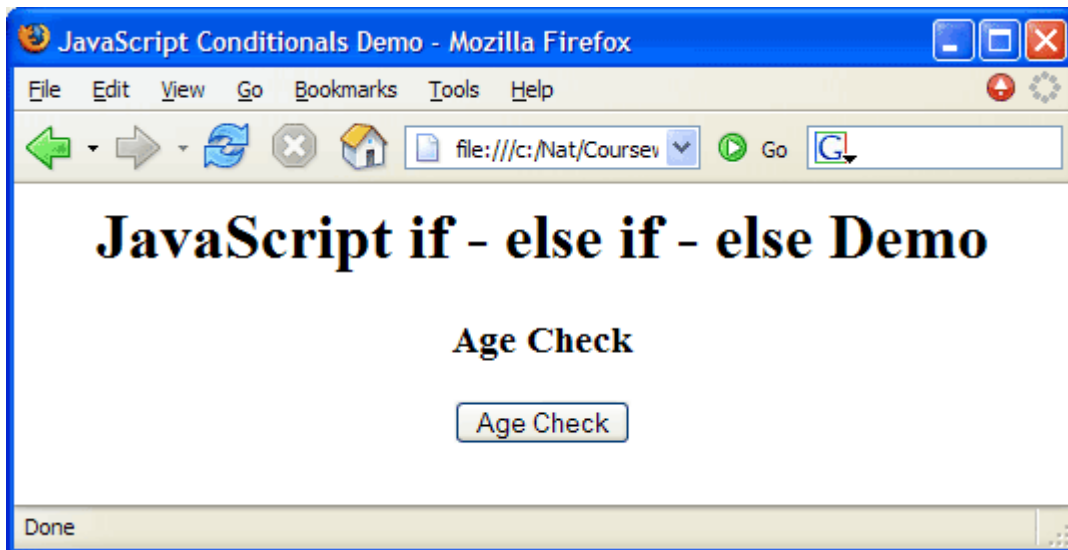
### *Code Sample: ConditionalsAndLoops/Demos/IfElseifElse.html*

```
<html>
<head>
<title>JavaScript Conditionals Demo</title>
<script type="text/javascript">
function checkAge() {
    var age = prompt("Your age?", "") || "";

    if (age >= 21) {
        alert("You can vote and drink!");
    } else if (age >= 18) {
        alert("You can vote, but can't drink.");
    } else {
        alert("You cannot vote or drink.");
    }
}
</script>
</head>
<body style="text-align:center">
<h1>JavaScript if - else if - else Demo</h1>
<h3>Age Check</h3>
<form>
    <input type="button" value="Age Check" onclick="checkAge ();">
</form>
</body>
</html>
```

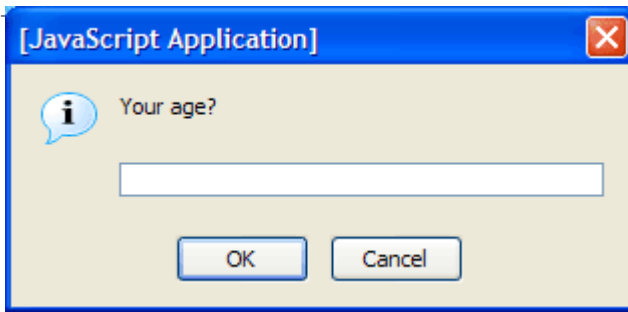
#### Code Explanation

The display of the page is shown below.



When the user clicks on the Age Check button, the following prompt pops up.

## JAVA SCRIPT



After the user enters his age, an alert pops up. The text of the alert depends on the user's age. The three possibilities are shown below.



### Compound Conditions

Compound conditions are conditions that check for multiple things. See the sample below.

```
if (age > 18 && isCitizen) {  
    alert("You can vote!");  
}  
  
if (age >= 16 && (isCitizen || hasGreenCard)) {  
    alert("You can work in the United States");  
}
```

### Short-circuiting

JavaScript is lazy (or efficient) about processing compound conditions. As soon as it can determine the overall result of the compound condition, it stops looking at the remaining parts of

## JAVA SCRIPT

the condition. This is useful for checking that a variable is of the right data type before you try to manipulate it.

To illustrate, take a look at the following sample.

### ***Code Sample: ConditionalsAndLoops/Demos/PasswordCheckBroken.html***

```
<html>
<head>
<title>Password Check</title>
<script type="text/javascript">
  var USER_PASS = prompt("Password:", ""); //ESC here causes problems
  var PASSWORD = "xyz";
</script>
</head>

<body>
<script type="text/javascript">
  if (USER_PASS.toLowerCase() == PASSWORD) {
    document.write("<h1>Welcome!</h1>");
  } else {
    document.write("<h1>Bad Password!</h1>");
  }
</script>
</body>
</html>
```

#### Code Explanation

Everything works fine as long as the user does what you expect. However, if the user clicks on the Cancel button when prompted for a password, the value null will be assigned to USER\_PASS. Because null is not a string, it does not have the toLowerCase() method. So the following line will result in a JavaScript error.

```
if (USER_PASS.toLowerCase() == password)
```

This can be fixed by using the typeof() function to first check if USER\_PASS is a string as shown in the sample below.

### ***Code Sample: ConditionalsAndLoops/Demos/PasswordCheck.html***

```
<html>
<head>
<title>Password Check</title>
<script type="text/javascript">
  var USER_PASS = prompt("Password:", "") || "";
  var PASSWORD = "xyz";
</script>
</head>

<body>
<script type="text/javascript">
  if (typeof USER_PASS == "string" && USER_PASS.toLowerCase() == PASSWORD) {
    document.write("<h1>Welcome!</h1>");
  } else {
    document.write("<h1>Bad Password!</h1>");
  }
</script>
</body>
</html>
```



## JAVA SCRIPT

```
</script>  
</body>  
</html>
```

### ***Switch / Case***

Syntax

```
switch (expression) {  
  case value :  
    statements;  
  case value :  
    statements;  
  default :  
    statements;  
}
```

Like if - else if - else statements, switch/case statements are used to run different code at different times. Generally, switch/case statements run faster than if - else if - else statements, but they are limited to checking for equality. Each case is checked to see if the *expression* matches the *value*.

Take a look at the following example.

### ***Code Sample: ConditionalsAndLoops/Demos/SwitchWithoutBreak.html***

```
<html>  
<head>  
<title>Switch</title>  
<script type="text/javascript">  
  var QUANTITY = 1;  
  switch (QUANTITY) {  
  case 1 :  
    alert("QUANTITY is 1");  
  case 2 :  
    alert("QUANTITY is 2");  
  default :  
    alert("QUANTITY is not 1 or 2");  
  }  
</script>  
</head>  
<body>  
  Nothing to show here.  
</body>  
</html>
```

Code Explanation

When you run this page in a browser, you'll see that all three alerts pop up, even though only the first case is a match. That's because if a match is found, none of the remaining cases is checked and all the remaining statements in the switch block are executed. To stop this process, you can insert a break statement, which will end the processing of the switch statement.

The corrected code is shown in the example below.

### ***Code Sample: ConditionalsAndLoops/Demos/SwitchWithBreak.html***

```
<html>  
<head>  
<title>Switch</title>
```

## JAVA SCRIPT

```
<script type="text/javascript">
var QUANTITY = 1;
switch (QUANTITY) {
case 1 :
  alert("QUANTITY is 1");
  break;
case 2 :
  alert("QUANTITY is 2");
  break;
default :
  alert("QUANTITY is not 1 or 2");
}
</script>
</head>
<body>
  Nothing to show here.
</body>
</html>
```

The following example shows how a switch/case statement can be used to record the user's browser type.

### ***Code Sample: ConditionalsAndLoops/Demos/BrowserSniffer.html***

```
<html>
<head>
<title>Simple Browser Sniffer</title>
<script type="text/javascript">
  switch (navigator.appName) {
  case "Microsoft Internet Explorer" :
    alert("This is IE!");
    break;
  case "Netscape" :
    alert("This is Mozilla!");
    break;
  default :
    alert("This is something other than IE or Mozilla!");
  }
</script>
</head>
<body>
  Nothing to show here.
</body>
</html>
```

#### Code Explanation

The navigator object, which is a child of window, holds information about the user's browser. In this case we are looking at the appName property, which has a value of "Microsoft Internet Explorer" for Internet Explorer and "Netscape" for Mozilla-based browsers (e.g, Navigator, Firefox).

### ***Exercise: Conditional Processing***

Duration: 20 to 30 minutes.

In this exercise, you will practice using conditional processing.

# JAVA SCRIPT

1. Open [ConditionalsAndLoops/Exercises/Conditionals.html](#) for editing.
2. Notice that there is an onload event handler that calls the greetUser() function. Create this function in the script block.
3. The function should do the following:
  1. Prompt the user for his/her gender and last name and store the results in variables.
  2. If the user enters a gender other than "Male" or "Female", prompt him/her to try again.
  3. If the user leaves the last name blank, prompt him/her to try again.
  4. If the user enters a number for the last name, tell him/her that a last name can't be a number and prompt him/her to try again.
  5. After collecting the gender and last name:
    - If the gender is valid, pop up an alert that greets the user appropriately (e.g, "Hello Ms. Smith!")
    - If the gender is not valid, pop up an alert that reads something like "XYZ is not a gender!"
4. Test your solution in a browser.
  1. Allow the user to enter his/her gender in any case.
  2. If the user enters a last name that does not start with a capital letter, prompt him/her to try again.

[Where is the solution?](#)

## Loops

There are several types of loops in JavaScript.

- while
- do...while
- for
- for...in

### *while Loop Syntax*

Syntax

```
while (conditions) {  
    statements;  
}
```

Something, usually a statement within the while block, must cause the condition to change so that it eventually becomes false and causes the loop to end. Otherwise, you get stuck in an infinite loop, which can bring down the browser.

### *do...while Loop Syntax*

Syntax

```
do {  
    statements;  
} while (conditions);
```

Again something, usually a statement within the do block, must cause the condition to change so that it eventually becomes false and causes the loop to end.

Unlike with while loops, the statements in do...while loops will always execute at least one time because the conditions are not checked until the end of each iteration.

## *for Loop Syntax*

Syntax

```
for (initialization; conditions; change) {  
  statements;  
}
```

In for loops, the initialization, conditions, and change are all placed up front and separated by semi-colons. This makes it easy to remember to include a change statement that will eventually cause the loop to end.

for loops are often used to iterate through arrays. The length property of an array can be used to check how many elements the array contains.

## *for...in Loop Syntax*

Syntax

```
for (var item in array) {  
  statements;  
}
```

for...in loops are specifically designed for looping through arrays. For reasons that will be better understood when we look at object augmentation, the above syntax has a slight flaw. If the Array class is changed, it is possible that the for...in loop includes more items than what you anticipated.

To be on the safe side, we suggest that you use a more verbose syntax as seen below.

Syntax

```
for (var item in array) if (array.hasOwnProperty(item)) {  
  statements;  
}
```

The hasOwnProperty() call will ensure that the item is indeed an element that you added to the array, not something that was inherited because of object augmentation.

## *Code Sample: ConditionalsAndLoops/Demos/Loops.html*

```
<html>  
<head>  
<title>JavaScript Loops</title>  
<script type="text/javascript">  
  var INDEX;  
  var BEATLES = [];  
  BEATLES["Guitar1"] = "John";  
  BEATLES["Bass"] = "Paul";  
  BEATLES["Guitar2"] = "George";  
  BEATLES["Drums"] = "Ringo";  
  var RAMONES = ["Joey", "Johnny", "Dee Dee", "Mark"];  
</script>  
</head>  
  
<body>  
<h1>JavaScript Loops</h1>  
<h2>while Loop</h2>
```

## JAVA SCRIPT

```
<script type="text/javascript">
INDEX = 0;
while (INDEX < 5) {
    document.write(INDEX);
    INDEX++;
}
</script>

<h2>do...while Loop</h2>
<script type="text/javascript">
INDEX = 1;
do {
    document.write(INDEX);
    INDEX++;
} while (INDEX < 5);
</script>

<h2>for Loop</h2>
<script type="text/javascript">
for (var INDEX=0; INDEX<=5; INDEX++) {
    document.write(INDEX);
}
</script>

<h2>for Loop with Arrays</h2>
<ol>
<script type="text/javascript">
for (var INDEX=0; INDEX<RAMONES.length; INDEX++) {
    document.write("<li>" + RAMONES[INDEX] + "</li>");
}
</script>
</ol>

<h2>for...in Loop</h2>
<ol>
<script type="text/javascript">
for (var instrument in BEATLES) if (BEATLES.hasOwnProperty(instrument)) {
    document.write("<li>Playing " + instrument + " there is " +
BEATLES[instrument] + "</li>");
}
</script>
</ol>
</body>
</html>
```

### Code Explanation

The sample above shows demos of all the aforementioned loops.

### ***Exercise: Working with Loops***

Duration: 20 to 30 minutes.

In this exercise, you will practice working with loops.

1. Open [ConditionalsAndLoops/Exercises/Loops.html](#) for editing. You will see that this file is similar to the solution from the last exercise.
2. Declare an additional variable called greeting.
3. Create an array called presidents that contains the last names of four or more past presidents.

## JAVA SCRIPT

---

4. Currently, the user only gets two tries to enter a valid gender and lastName. Modify the code so that, in both cases, the user continues to get prompted until he enters valid data.
  5. Change the switch block so that it assigns an appropriate value (e.g, "Hello Ms. Smith") to the greeting variable rather than popping up an alert.
  6. After the switch block, write code that alerts the user by name if he has the same last name as a president. There is no need to alert those people who have non-presidential names.
- 
1. Modify the code so that the first prompt for gender reads "What gender are you: Male or Female?", but all subsequent prompts for gender read "You must enter 'Male' or 'Female'. Try again:".
  2. Modify the code so that the first prompt for last name reads "Enter your last name:", but all subsequent prompts for last name read "Please enter a valid last name:".
  3. If the user presses the *Cancel* button on a prompt dialog, it returns null. Test this. It very likely results in a JavaScript error. If so, fix the code so that no JavaScript error occurs.
  4. For those people who do not have presidential names, pop up an alert that tells them their names are not presidential.

[Where is the solution?](#)

## Conditionals and Loops Conclusion

In this lesson of the JavaScript tutorial, you learned to work with JavaScript if-else if-else and switch/case conditionals and several types of loops.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## JavaScript Form Validation

In this lesson of the JavaScript tutorial, you will learn...

---

## JAVA SCRIPT

---

1. To access data entered by users in forms.
2. To validate text fields and passwords.
3. To validate radio buttons.
4. To validate checkboxes.
5. To validate select menus.
6. To validate textareas.
7. To write clean, reusable validation functions.
8. To catch focus, blur, and change events.

### Accessing Form Data

All forms on a web page are stored in the `document.forms[]` array. 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.

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

#### Syntax

```
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.

#### *Code Sample: FormValidation/Demos/FormFields.html*

```
<html>
<head>
<title>Form Fields</title>
<script type="text/javascript">
  function changeBg() {
    var userName = document.forms[0].UserName.value;
    var bgColor = document.BgForm.BgColor.value;

    document.bgColor = bgColor;
    alert(userName + ", the background color is " + bgColor + ".");
  }
</script>
</head>
<body>
<h1>Change Background Color</h1>
<form name="BgForm">

  Your Name: <input type="text" name="UserName" size="10"><br/>
  Background Color: <input type="text" name="BgColor" size="10"><br/>

  <input type="button" value="Change Background" onclick="changeBg();" />
</form>
</body>
</html>
```

#### Code Explanation

Some things to notice:

---

## JAVA SCRIPT

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

### ***Exercise: Textfield to Textfield***

Duration: 15 to 25 minutes.

In this exercise, you will write a function that bases the value of one text field on the value of another.

1. Open FormValidation/Exercises/TextfieldToTextField.html for editing.
2. Write a function called getMonth() that passes the month number entered by the user to the monthAsString() function in DateUDFs.js and writes the result in the MonthName field.

### ***Code Sample: FormValidation/Exercises/TextfieldToTextField.html***

```
<html>
<head>
<title>Textfield to Textfield</title>
<script src="DateUDFs.js" type="text/javascript"></script>
<script type="text/javascript">
  /*
   Write a function called getMonth() that passes the
   month number entered by the user to the monthAsString()
   function in DateUDFs.js and writes the result in
   the MonthName field.
  */
</script>
</head>
<body>
<h1>Month Check</h1>
<form name="DateForm">
  Month Number: <input type="text" name="MonthNumber" size="2">
  <input type="button" value="Get Month" onclick="getMonth();" ><br>
  Month Name: <input type="text" name="MonthName" size="10">
</form>
</body>
</html>
```

1. If the user enters a number less than 1 or greater than 12 or a non-number, have the function write "Bad Number" in the MonthName field.
2. If the user enters a decimal between 1 and 12 (inclusive), strip the decimal portion of the number.

[Where is the solution?](#)

## **Basics of Form Validation**

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. For example, the following form will never be submitted.



## JAVA SCRIPT

```
<form action="Process.html" onsubmit="return false;">
  <!--Code for form fields would go here-->
  <input type="submit" value="Submit Form">
</form>
```

Of course, when validating a form, we only want the form *not* to submit if something is wrong. The trick is to return false if there is an error, but otherwise return true. So instead of returning false, we call a validation function, which will specify the result to return.

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

### ***The this Object***

Notice that we pass the validate() function the this object. The this object refers to the current object - whatever object (or element) the this keyword appears in. In the case above, the this object refers to the form object. So the entire form object is passed to the validate() function. Let's take a look at a simple example.

### ***Code Sample: FormValidation/Demos/Login.html***

```
<html>
<head>
<title>Login</title>
<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>
</head>
<body>
<h1>Login Form</h1>
<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">
</p>
</form>
</body>
</html>
```

Code Explanation

## JAVA SCRIPT

1. When the user submits the form, the onsubmit event handler captures the event and calls the validate() function, passing in the form object.
2. The validate() function stores the form object in the form variable.
3. The values entered into the Username and Password fields are stored in variables (userName and password).
4. An if condition is used to check if userName is an empty string. If it is, an alert pops up explaining the problem and the function returns false. The function stops processing and the form does *not* submit.
5. An if condition is used to check that password is an empty string. If it is, an alert pops up explaining the problem and the function returns false. The function stops processing and the form does *not* submit.
6. If neither if condition catches a problem, the function returns true and the form submits.

### ***Cleaner Validation***

There are a few improvements we can make on the last example.

One problem is that the validation() 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. Why not just tell the user all the mistakes that need to be corrected, so he doesn't have to keep submitting the form to find each subsequent error?

Another problem is that the code is not written in a way that makes it easily reusable. For example, checking for the length of user-entered values is a common thing to do, so it would be nice to have a ready-made function to handle this.

These improvements are made in the example below.

### ***Code Sample: FormValidation/Demos/Login2.html***

```
<html>
<head>
<title>Login</title>
<script type="text/javascript">
function validate(form) {
    var userName = form.Username.value;
    var password = form.Password.value;
    var errors = [];

    if (!checkLength(userName)) {
        errors[errors.length] = "You must enter a username.";
    }

    if (!checkLength(password)) {
        errors[errors.length] = "You must enter a password.";
    }

    if (errors.length > 0) {
        reportErrors(errors);
        return false;
    }

    return true;
}
```

## JAVA SCRIPT

```
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>
</head>
<body>
<h1>Login Form</h1>
<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">
</p>
</form>
</body>
</html>
```

### Code Explanation

Some things to notice:

1. Two additional functions are created: checkLength() and reportErrors().
  - The checkLength() function takes three arguments, the text to examine, the required minimum length, and the required maximum length. If the minimum length and maximum length are not passed, defaults of 1 and 10000 are used.
  - The reportErrors() function takes one argument, an array holding the errors. It loops through the errors array creating an error message and then it pops up an alert with this message. The \n is an escape character for a newline.
2. In the main validate() function, a new array, errors, is created to hold any errors that are found.
3. userName and password are passed to checkLength() for validation.
  - If errors are found, they are appended to errors.
4. If there are any errors in errors (i.e, if its length is greater than zero), then errors is passed to reportErrors(), which pops up an alert letting the user know where the errors are. The validate() function then returns false and the form is *not* submitted.
5. If no errors are found, the validate() function returns true and the form is submitted.

## JAVA SCRIPT

By modularizing the code in this way, it makes it easy to add new validation functions. In the next examples we will move the reusable validation functions into a separate JavaScript file called `FormValidation.js`.

### ***Exercise: Validating a Registration Form***

Duration: 25 to 40 minutes.

In this exercise, you will write code to validate a registration form.

1. Open [FormValidation/Exercises/FormValidation.js](#) for editing.
  - Create a function called `compareValues()` that takes two arguments: `val1` and `val2`. The function should return:
    - 0 if the two values are equal
    - -1 if `val1` is greater than `val2`
    - 1 if `val2` is greater than `val1`
  - Create a function called `checkEmail()` that takes one argument: `email`. The function should return:
    - false if email has fewer than 6 characters
    - false if email does not contain an `@` symbol
    - false if email does not contain a period (`.`)
    - true otherwise
2. Open [FormValidation/Exercises/Register.html](#) for editing.
  - Add code so that the functions in [FormValidation.js](#) are accessible from this page.
  - Create a `validate()` function that does the following:
    - Checks that the `FirstName`, `LastName`, `City`, `Country`, `UserName`, and `Password1` fields are filled out.
    - Checks that the middle initial is a single character.
    - Checks that the state is exactly two characters.
    - Checks that the email is a valid email address.
    - Checks that the values entered into `Password1` and `Password2` are the same.
    - If there are errors, passes `reportErrors()` the errors array and returns false.
    - If there are no errors, returns true.
3. Test your solution in a browser.

In [FormValidation/Exercises/FormValidation.js](#), modify the `checkEmail()` function so that it also checks to see that the final period (`.`) is after the final `@` symbol. The solution is included in [FormValidation/Solutions/FormValidation.js](#).

[Where is the solution?](#)

## Validating Radio Buttons

Radio buttons that have the same name are grouped as arrays. Generally, the goal in validating a radio button array is to make sure that the user has checked one of the options. Individual radio buttons have the `checked` property, which is true if the button is checked and false if it is not. The example below shows a simple function for checking radio button arrays.

### ***Code Sample: FormValidation/Demos/RadioArrays.html***

```
<html>  
<head>
```

## JAVA SCRIPT

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

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>
</head>
<body>
<h1>Ice Cream Form</h1>
<form method="post" action="Process.html"
  onsubmit="return validate(this);">
  <b>Cup or Cone?</b>
  <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>

</body>
</html>
```

### Code Explanation

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. Since it is only possible for one option to be checked, there is no reason to continue looking once a checked button has been found. If none of the buttons is checked, the function returns false.

### Validating Checkboxes

Like radio buttons, checkboxes have the checked property, which is true if the button is checked and false if it is not. However, unlike radio buttons, checkboxes are not stored as arrays. The example below shows a simple function for checking to make sure a checkbox is checked.

#### *Code Sample: FormValidation/Demos/CheckBoxes.html*

```
<html>
<head>
<title>Checkboxes</title>
<script type="text/javascript">
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;
}

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>
</head>
<body>
<h1>Ice Cream Form</h1>
<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>

</body>
</html>
```

## Validating Select Menus

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 in the example below makes sure that the first option is not selected.

### *Code Sample: FormValidation/Demos/SelectMenus.html*

```
<html>
<head>
<title>Check Boxes</title>
<script type="text/javascript">
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);
}

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>
</head>
<body>
<h1>Ice Cream Form</h1>
<form method="post" action="Process.html"
  onsubmit="return validate(this);">
  <b>Flavor:</b>
  <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>
```

```
</body>  
</html>
```

## Focus, Blur, and Change Events

*Focus*, *blur* and *change* events can be used to improve the user experience.

### ***Focus and Blur***

Focus and blur events are caught with the `onfocus` and `onblur` event handlers. These events have corresponding `focus()` and `blur()` methods. The example below shows

1. how to set focus on a field.
2. how to capture when a user leaves a field.
3. how to prevent focus on a field.

### ***Code Sample: FormValidation/Demos/FocusAndBlur.html***

```
<html>  
<head>  
<title>Focus and Blur</title>  
<script src="DateUDFs.js" type="text/javascript"></script>  
<script type="text/javascript">  
  function getMonth() {  
    var elemMonthNumber = document.DateForm.MonthNumber;  
    var monthNumber = elemMonthNumber.value;  
  
    var elemMonthName = document.DateForm.MonthName;  
    var month = monthAsString(elemMonthNumber.value);  
  
    elemMonthName.value = (monthNumber > 0 && monthNumber <=12) ? month : "Bad  
Number";  
  }  
</script>  
</head>  
<body onload="document.DateForm.MonthNumber.focus();">  
<h1>Month Check</h1>  
<form name="DateForm">  
  Month Number:  
  <input type="text" name="MonthNumber" size="2" onblur="getMonth();">  
  Month Name:  
  <input type="text" name="MonthName" size="10" onfocus="this.blur();">  
</form>  
</body>  
</html>
```

#### Code Explanation

#### Things to notice:

1. When the document is loaded, the `focus()` method of the text field element is used to set focus on the `MonthNumber` element.
2. When focus leaves the `MonthNumber` field, the `onblur` event handler captures the event and calls the `getMonth()` function.
3. The `onfocus` event handler of the `MonthName` element triggers a call to the `blur()` method of this (the `MonthName` element itself) to prevent the user from focusing on the `MonthName` element.



## Change

Change events are caught when the value of a text element changes or when the selected index of a select element changes. The example below shows how to capture a change event.

### Code Sample: FormValidation/Demos/Change.html

```
<html>
<head>
<title>Change</title>
<script src="DateUDFs.js" type="text/javascript"></script>
<script type="text/javascript">
  function getMonth() {
    var elemMonthNumber = document.DateForm.MonthNumber;
    var i = elemMonthNumber.selectedIndex;
    var monthNumber = elemMonthNumber[i].value;

    var elemMonthName = document.DateForm.MonthName;
    var month = monthAsString(monthNumber);

    elemMonthName.value = (i === 0) ? "" : month;
  }
</script>
</head>
<body onload="document.DateForm.MonthNumber.focus();" >
<h1>Month Check</h1>
<form name="DateForm">
  Month Number:
  <select name="MonthNumber" onchange="getMonth();" >
    <option>--Choose--</option>
    <option value="1">1</option>
    <option value="2">2</option>
    <option value="3">3</option>
    <option value="4">4</option>
    <option value="5">5</option>
    <option value="6">6</option>
    <option value="7">7</option>
    <option value="8">8</option>
    <option value="9">9</option>
    <option value="10">10</option>
    <option value="11">11</option>
    <option value="12">12</option>
  </select><br>
  Month Name: <input type="text" name="MonthName" size="10"
    onfocus="this.blur();" >
</form>
</body>
</html>
```

#### Code Explanation

This is similar to the last example. The only major difference is that MonthNumber is a select menu instead of a text field and that the getMonth() function is called when a different option is selected.

### Validating Textareas

Textareas can be validated the same way that text fields are by using the `checkLength()` function shown earlier. However, because textareas generally allow for many more characters, it's often difficult for the user to know if he's exceeded the limit. It could be helpful to let the user know if there's a problem as soon as focus leaves the textarea. The example below, which contains a more complete form for ordering ice cream, includes a function that alerts the user if there are too many characters in a textarea.

#### ***Code Sample: FormValidation/Demos/IceCreamForm.html***

```
<html>
<head>
<title>Check Boxes</title>
<script type="text/javascript">
function validate(form){
  var errors = [];

  if ( !checkRadioArray(form.container) ) {
    errors[errors.length] = "You must choose a cup or cone.";
  }

  if ( !checkCheckBox(form.terms) ) {
    errors[errors.length] = "You must agree to the terms.";
  }

  if ( !checkSelect(form.flavor) ) {
    errors[errors.length] = "You must choose a flavor.";
  }

  if (errors.length > 0) {
    reportErrors(errors);
    return false;
  }

  return true;
}

function checkRadioArray(radioButton){
  for (var i=0; i < radioButton.length; i++) {
    if (radioButton[i].checked) {
      return true;
    }
  }
  return false;
}

function checkCheckBox(cb){
  return cb.checked;
}

function checkSelect(select){
  return (select.selectedIndex > 0);
}

function checkLength(text, min, max){
  min = min || 1;
```

## JAVA SCRIPT

```
max = max || 10000;
if (text.length < min || text.length > max) {
    return false;
}
return true;
}

function checkTextArea(textArea, max){
    var numChars, chopped, message;
    if (!checkLength(textArea.value, 0, max)) {
        numChars = textArea.value.length;
        chopped = textArea.value.substr(0, max);
        message = 'You typed ' + numChars + ' characters.\n';
        message += 'The limit is ' + max + '.';
        message += 'Your entry will be shortened to:\n\n' + chopped;
        alert(message);
        textArea.value = chopped;
    }
}

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>
</head>
<body>
<h1>Ice Cream Form</h1>
<form method="post" action="Process.html" onsubmit="return validate(this);">
    <p><b>Cup or Cone?</b>
        <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
    </p>
    <p>
        <b>Flavor:</b>
        <select name="flavor">
            <option value="0" selected></option>
            <option value="choc">Chocolate</option>
            <option value="straw">Strawberry</option>
            <option value="van">Vanilla</option>
        </select>
    </p>
    <p>
        <b>Special Requests:</b><br>
        <textarea name="requests" cols="40" rows="6" wrap="virtual"
            onblur="checkTextArea(this, 100);"></textarea>
    </p>
    <p>
        <input type="checkbox" name="terms">
        I understand that I'm really not going to get any ice cream.
    </p>
    <input type="submit" value="Place Order">
</form>
</body>
</html>
```

## JAVA SCRIPT

```
</form>  
  
</body>  
</html>
```

### ***Exercise: Improving the Registration Form***

Duration: 15 to 25 minutes.

In this exercise, you will make some improvements to the registration form from the last exercise.

1. Open [FormValidation/Exercises/FormValidation2.js](#) in your editor. You will see that the functions discussed above have been added: `checkRadioArray()`, `checkCheckBox()`, `checkSelect()`, and `checkTextArea()`.
2. Open [FormValidation/Exercises/Register2.html](#) for editing.
  - Notice that the following changes have been made to the form:
    - State has been changed from a textfield to a select menu. The first option is meaningless. The next 51 options are U.S. states. The rest of the options are Canadian provinces.
    - Country has been changed to a radio array.
    - A Comments field has been added.
    - A Terms checkbox has been added.
  - Write code that:
    - Checks that a country is selected.
    - Checks that the country and state selection are in sync.
    - Checks that the terms have been accepted.
  - Add an event handler to the Comments textarea that alerts the user if the comment is too long.
3. Test your solution in a browser.

[Where is the solution?](#)

## JavaScript Form Validation Conclusion

In this lesson of the JavaScript tutorial, you have learned to capture form events, to reference form fields, and to write clean, reusable form validation scripts.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## Working with Images

**In this lesson of the JavaScript tutorial, you will learn...**

1. To create image rollovers.
2. To create backward-compatible image rollovers.
3. To preload images.
4. To create a slide show.

## Image Rollovers

Image rollovers are commonly used to create a more interesting user experience and to help highlight navigation points. When the user hovers the mouse over an image, the source of the image is modified, so that a different image appears. When the user hovers the mouse back off the image, the original source is restored. The code below shows how to create a simple rollover.

### *Code Sample: DynamicImages/Demos/SimpleRollover.html*

```
<html>
<head>
<title>Simple Image Rollover</title>
</head>
<body>
<div style="text-align:center;">
<h1>Simple Image Rollover</h1>

<p>Who are you calling simple?</p>
</div>
</body>
</html>
```

#### Code Explanation

The mouse-over event is captured with the img tag's onmouseover event handler. When this happens, the following JavaScript code is called.

```
this.src = 'Images/Hulk.jpg';
```

The this object refers to the current object - whatever object (or element) the this keyword appears in. In the case above, the this object refers to the img object, which has a property called src that holds the path to the image. The code above sets the src to "[Images/Hulk.jpg](#)".

Likewise, the mouse-out event is captured with the img tag's onmouseout event handler. When this happens, the following JavaScript code is called.

```
this.src = 'Images/Banner.jpg';
```

This code sets the src to "[Images/Banner.jpg](#)," which is what it originally was.

### *Backward Compatibility*

The code above should work fine in Firefox, Internet Explorer 4.0 and later, and Netscape 6 and later. However, in earlier versions of browsers, images could not capture mouse events. The workaround is to wrap the <img> tag in an <a> tag and to put the event handlers in the <a> tag as shown below.

### *Code Sample: DynamicImages/Demos/SimpleRollover-backward.html*

```
<html>
<head>
<title>Simple Image Rollover</title>
```

## JAVA SCRIPT

```
</head>
<body>
<div style="text-align:center;">
<h1>Simple Image Rollover</h1>
<a href="#"
  onmouseover="document.MyImage.src = 'Images/Hulk.jpg';"
  onmouseout="document.MyImage.src = 'Images/Banner.jpg';">
</a>
<p>Who are you calling simple?</p>
</div>
</body>
</html>
```

### *An Image Rollover Function*

Image rollovers can be handled by a function as well. The two examples below show an image rollover function for modern browsers and a backward-compatible image rollover function.

### *Code Sample: DynamicImages/Demos/SimpleRolloverFunction.html*

```
<html>
<head>
<title>Simple Image Rollover Function</title>
<script type="text/javascript">
function imageRollover(img, imgSrc){
  img.src = imgSrc;
}
</script>
</head>
<body>
<div style="text-align:center;">
<h1>Simple Image Rollover Function</h1>


<p>Who are you calling simple?</p>
</div>
</body>
</html>
```

### *Code Sample: DynamicImages/Demos/SimpleRolloverFunction-backward.html*

```
<html>
<head>
<title>Simple Image Rollover Function</title>
<script type="text/javascript">
function imageRollover(imgName, imgSrc){
  if (document.images) {
    document.images[imgName].src = imgSrc;
  }
}
</script>
</head>
<body>
```

## JAVA SCRIPT

```
<div style="text-align:center;">
<h1>Simple Image Rollover Function</h1>
<a href="#"
  onmouseover="imageRollover('MyImage', 'Images/Hulk.jpg');"
  onmouseout="imageRollover('MyImage', 'Images/Banner.jpg');">
</a>
<p>Who are you calling simple?</p>
</div>
</body>
</html>
```

### Code Explanation

Why the check for document.images? Some early versions of browsers don't support the images array.

## Preloading Images

When working with files on a local machine, image rollovers like the ones we have seen in previous examples work just fine. However, when the user first hovers over an image rollover image, the new image file has to be found and delivered to the page. If the new image is on a far-away server, this can take a few moments, causing an ugly pause in the rollover effect. This can be prevented by preloading images.

Images can be preloaded by creating an Image object with JavaScript and assigning a value to the src of that Image. A sample is shown below.

### *Code Sample: DynamicImages/Demos/PreloadingImages.html*

```
<html>
<head>
<title>Preloading Images</title>
<script type="text/javascript">

var IMAGE_PATHS = [];
IMAGE_PATHS[0] = "Images/Hulk.jpg";
IMAGE_PATHS[1] = "Images/Batman.jpg";

var IMAGE_CACHE = [];

for (var i=0; i<IMAGE_PATHS.length; i++) {
  IMAGE_CACHE[i] = new Image();
  IMAGE_CACHE[i].src = IMAGE_PATHS[i];
}

function imageRollover(img, imgSrc) {
  img.src = imgSrc;
}
</script>
</head>
<body>
<div style="text-align:center;">
<h1>Simple Image Rollover Function</h1>

Who are you calling simple?</p>
</div>
</body>
</html>
```

### Code Explanation

Notice that the code is not in a function. It starts working immediately as follows:

1. An array called `IMAGE_PATHS` is created to hold the paths to the images that need to be preloaded.

```
var IMAGE_PATHS = [];
```

2. An array element is added for each image to be preloaded.

```
3. IMAGE_PATHS[0] = "Images/Hulk.jpg";
   IMAGE_PATHS[1] = "Images/Batman.jpg";
```

4. An array called `IMAGE_CACHE` is created to hold the `Image` objects that will hold the preloaded images.

```
var IMAGE_CACHE = [];
```

5. A for loop is used to create an `Image` object and load in an image for each image path in `IMAGE_PATHS`.

```
6. for (var i=0; i<IMAGE_PATHS.length; i++) {
7.   IMAGE_CACHE[i]=new Image();
8.   IMAGE_CACHE[i].src=IMAGE_PATHS[i];
   }
```

### ***Exercise: Creating a Slide Show***

Duration: 20 to 30 minutes.

In this exercise, you will practice working with images by creating a slide show.

1. Open [DynamicImages/Exercises/SlideShow.html](#) for editing.
2. Write code to preload [Images/Banner.jpg](#), [Images/Hulk.jpg](#), [Images/Bruce.jpg](#), and [Images/Batman.jpg](#). The paths to the images should be stored in an array called `IMAGE_PATHS`. The `Image` objects should be stored in an array called `IMAGE_CACHE`.
3. Create a function called `changeSlide()` that takes one parameter: `dir`, and behaves as follows:
  - If `dir` equals 1, it changes the slide to the next image as stored in the `IMAGE_PATHS` array.
  - If `dir` equals -1, it changes the slide to the previous image as stored in the `IMAGE_PATHS` array.
  - If the user is viewing the last slide and clicks the Next button, the first slide should appear.
  - If the user is viewing the first slide and clicks the Previous button, the last slide should appear.
4. Test your solution in a browser.

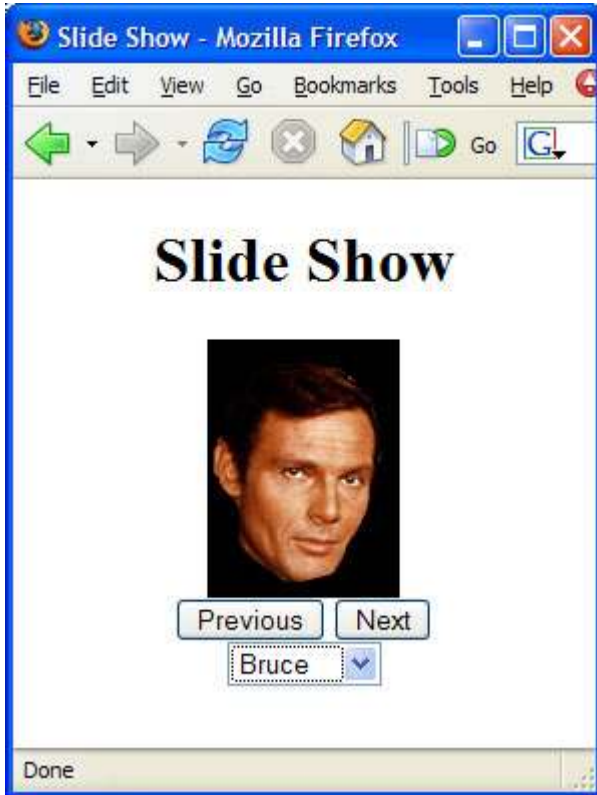
1. Add a dropdown menu below the Previous and Next buttons that contains the names of the images without their extensions: "Banner", "Hulk", "Bruce" and "Batman".



## JAVA SCRIPT

2. When the user selects an image from the dropdown, have that image appear above.
3. When the user changes the image above using the Previous and Next buttons, have the dropdown menu keep in sync (i.e, show the name of the current image).

The solution should look like the screenshot below.



[Where is the solution?](#)

### Working with Images Conclusion

In this lesson of the JavaScript tutorial, you have learned how JavaScript can be used to manipulate HTML images to create image rollover effects and slide shows.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## Regular Expressions

In this lesson of the JavaScript tutorial, you will learn...

1. To use regular expressions for advanced form validation.
2. To use regular expressions and backreferences to clean up form entries.

## Getting Started

Regular expressions are used to do sophisticated pattern matching, which can often be helpful in form validation. For example, a regular expression can be used to check whether an email address entered into a form field is syntactically correct. JavaScript supports Perl-compatible regular expressions.

There are two ways to create a regular expression in JavaScript:

1. Using literal syntax

## JAVA SCRIPT

---

```
var reExample = /pattern/;
```

### 2. Using the RegExp() constructor

```
var reExample = new RegExp("pattern");
```

Assuming you know the regular expression pattern you are going to use, there is no real difference between the two; however, if you don't know the pattern ahead of time (e.g. you're retrieving it from a form), it can be easier to use the RegExp() constructor.

## ***JavaScript's Regular Expression Methods***

The regular expression method in JavaScript has two main methods for testing strings: test() and exec().

### **The exec() Method**

The exec() method takes one argument, a string, and checks whether that string contains *one or more* matches of the pattern specified by the regular expression. If one or more matches is found, the method returns a result array with the starting points of the matches. If no match is found, the method returns null.

### **The test() Method**

The test() method also takes one argument, a string, and checks whether that string contains a match of the pattern specified by the regular expression. It returns true if it does contain a match and false if it does not. This method is very useful in form validation scripts. The code sample below shows how it can be used for checking a social security number. Don't worry about the syntax of the regular expression itself. We'll cover that shortly.

### ***Code Sample: RegularExpressions/Demos/SsnChecker.html***

```
<html>
<head>
<title>ssn Checker</title>
<script type="text/javascript">
var RE_SSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;

function checkSsn(ssn) {
  if (RE_SSN.test(ssn)) {
    alert("VALID SSN");
   } else {
    alert("INVALID SSN");
   }
}
</script>
</head>
<body>
<form onsubmit="return false;">
  <input type="text" name="ssn" size="20">
  <input type="button" value="Check"
    onclick="checkSsn(this.form.ssn.value);">
</form>
```

# JAVA SCRIPT

```
</body>
```

```
</html>
```

Code Explanation

Let's examine the code more closely:

1. First, a variable containing a regular expression object for a social security number is declared.

```
var RE_SSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;
```

2. Next, a function called `checkSsn()` is created. This function takes one argument: `ssn`, which is a string. The function then tests to see if the string matches the regular expression pattern by passing it to the regular expression object's `test()` method. If it does match, the function alerts "VALID SSN". Otherwise, it alerts "INVALID SSN".

```
3. function checkSsn(ssn) {  
4.   if (RE_SSN.test(ssn)) {  
5.     alert("VALID SSN");  
6.   } else {  
7.     alert("INVALID SSN");  
8.   }  
}
```

9. A form in the body of the page provides a text field for inserting a social security number and a button that passes the user-entered social security number to the `checkSsn()` function.

```
10. <form onsubmit="return false;">  
11.   <input type="text" name="ssn" size="20">  
12.   <input type="button" value="Check"  
13.     onclick="checkSsn(this.form.ssn.value);">  
</form>
```

## Flags

Flags appearing after the end slash modify how a regular expression works.

- The `i` flag makes a regular expression case insensitive. For example, `/aeiou/i` matches all lowercase and uppercase vowels.
- The `g` flag specifies a global match, meaning that all matches of the specified pattern should be returned.

## String Methods

There are several String methods that use regular expressions.

### The `search()` Method

The `search()` method takes one argument: a regular expression. It returns the index of the first character of the substring matching the regular expression. If no match is found, the method returns `-1`.

```
"Webucator".search(/cat/); //returns 4
```

## The split() Method

The split() method takes one argument: a regular expression. It uses the regular expression as a delimiter to split the string into an array of strings.

```
"Webucator".split(/[aeiou]/);  
/*  
 returns an array with the following values:  
 "W", "b", "c", "t", "r"  
*/
```

## The replace() Method

The replace() method takes two arguments: a regular expression and a string. It replaces the first regular expression match with the string. If the g flag is used in the regular expression, it replaces all matches with the string.

```
"Webucator".replace(/cat/, "dog"); //returns Webudogor  
"Webucator".replace(/[aeiou]/g, "x"); //returns Wxbxcxtxr
```

## The match() Method

The match() method takes one argument: a regular expression. It returns each substring that matches the regular expression pattern.

```
"Webucator".match(/[aeiou]/g);  
/*  
 returns an array with the following values:  
 "e", "u", "a", "o"  
*/
```

## Regular Expression Syntax

A regular expression is a pattern that specifies a list of characters. In this section, we will look at how those characters are specified.

### ***Start and End ( ^ \$ )***

A caret (^) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern ^foo can be found in "food", but not in "barfood".

A dollar sign (\$) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern foo\$ can be found in "curfoo", but not in "food".

## JAVA SCRIPT

---

### ***Number of Occurrences ( ? + \* {} )***

The following symbols affect the number of occurrences of the preceding character: ?, +, \*, and {}.

A questionmark (?) indicates that the preceding character should appear zero or one times in the pattern.

- The pattern foo? can be found in "food" and "fod", but not "faod".

A plus sign (+) indicates that the preceding character should appear one or more times in the pattern.

- The pattern fo+ can be found in "fod", "food" and "foood", but not "fd".

A asterisk (\*) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern fo\*d can be found in "fd", "fod" and "food".

Curly brackets with one parameter ( {n} ) indicate that the preceding character should appear exactly n times in the pattern.

- The pattern fo{3}d can be found in "foood" , but not "food" or "foood".

Curly brackets with two parameters ( {n1,n2} ) indicate that the preceding character should appear between n1 and n2 times in the pattern.

- The pattern fo{2,4}d can be found in "food","foood" and "foood", but not "fod" or "fooooo".

Curly brackets with one parameter and an empty second parameter ( {n,} ) indicate that the preceding character should appear at least n times in the pattern.

- The pattern fo{2,}d can be found in "food" and "fooooo", but not "fod".

### ***Common Characters ( . \d \D \w \W \s \S )***

A period ( . ) represents any character except a newline.

- The pattern fo.d can be found in "food", "foad", "fo9d", and "fo\*d".

Backslash-d ( \d ) represents any digit. It is the equivalent of [\[0-9\]](#).

- The pattern fo\dd can be found in "fo1d", "fo4d" and "fo0d", but not in "food" or "fodd".

Backslash-D ( \D ) represents any character except a digit. It is the equivalent of [\[^0-9\]](#).

- The pattern fo\Dd can be found in "food" and "foad", but not in "fo4d".

## JAVA SCRIPT

---

Backslash-w ( \w ) represents any word character (letters, digits, and the underscore ( \_ )).

- The pattern fo\wd can be found in "food", "fo\_d" and "fo4d", but not in "fo\*d".

Backslash-W ( \W ) represents any character except a word character.

- The pattern fo\Wd can be found in "fo\*d", "fo@d" and "fo.d", but not in "food".

Backslash-s ( \s ) represents any whitespace character (e.g, space, tab, newline, etc.).

- The pattern fo\s d can be found in "fo d", but not in "food".

Backslash-S ( \S ) represents any character except a whitespace character.

- The pattern fo\Sd can be found in "fo\*d", "food" and "fo4d", but not in "fo d".

### **Grouping ( [ ] )**

Square brackets ( [ ] ) are used to group options.

- The pattern f[aeiou]d can be found in "fad" and "fed", but not in "food", "faed" or "fd".
- The pattern f[aeiou]{2}d can be found in "faed" and "feod", but not in "fod", "fed" or "fd".

### **Negation ( ^ )**

When used after the first character of the regular expression, the caret ( ^ ) is used for negation.

- The pattern f[^aeiou]d can be found in "fqd" and "f4d", but not in "fad" or "fed".

### **Subpatterns ( () )**

Parentheses ( () ) are used to capture subpatterns.

- The pattern f(oo)?d can be found in "food" and "fd", but not in "fod".

### **Alternatives ( | )**

The pipe ( | ) is used to create optional patterns.

- The pattern foo\$|^bar can be found in "foo" and "bar", but not "foobar".

### **Escape Character ( \ )**

The backslash ( \ ) is used to escape special characters.

- The pattern fo\d can be found in "fo.d", but not in "food" or "fo4d".

## Backreferences

Backreferences are special wildcards that refer back to a subpattern within a pattern. They can be used to make sure that two subpatterns match. The first subpattern in a pattern is referenced as `\1`, the second is referenced as `\2`, and so on.

For example, the pattern `([bmpw])o\1` matches `â€œbobâ€`, `â€œmomâ€`, `â€œpopâ€`, and `â€œwowâ€`, but not "bop" or "pow".

A more practical example has to do matching the delimiter in social security numbers. Examine the following regular expression.

```
^\d{3}([\ -]?)\d{2}([\ -]?)\d{4}$
```

Within the caret (^) and dollar sign (\$), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. This pattern will be matched in all of following strings (and more).

- 123-45-6789
- 123 45 6789
- 123456789
- 123-45 6789
- 123 45-6789
- 123-456789

The last three strings are not ideal, but they do match the pattern. Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this.

```
^\d{3}([\ -]?)\d{2}\1\d{4}$
```

The `\1` refers back to the first subpattern. Only the first three strings listed above match this regular expression.

## Form Validation with Regular Expressions

Regular expressions make it easy to create powerful form validation functions. Take a look at the following example.

### *Code Sample: RegularExpressions/Demos/Login.html*

```
<html>
<head>
<title>Login</title>
<script type="text/javascript">

var RE_EMAIL = /^(\\w+[\\-\\.])*\\w+@[\\w\\.]+[A-Za-z]+$/;
var RE_PASSWORD = /^[A-Za-z\\d]{6,8}$/;

function validate(form){
  var email = form.Email.value;
```



## JAVA SCRIPT

```
var password = form.Password.value;
var errors = [];

if (!RE_EMAIL.test(email)) {
    errors[errors.length] = "You must enter a valid email address.";
}

if (!RE_PASSWORD.test(password)) {
    errors[errors.length] = "You must enter a valid password.";
}

if (errors.length > 0) {
    reportErrors(errors);
    return false;
}

return true;
}

function reportErrors(errors){
    var msg = "There were some problems...\n";
    for (var i = 0; i<errors.length; i++) {
        var numError = i + 1;
        msg += "\n" + numError + ". " + errors[i];
    }
    alert(msg);
}
</script>
</head>
<body>
<h1>Login Form</h1>
<form method="post" action="Process.html" onsubmit="return validate(this);">

    Email: <input type="text" name="Email" size="25"><br/>
    Password: <input type="password" name="Password" size="10"><br/>
    *Password must be between 6 and 10 characters and
    can only contain letters and digits.<br/>

    <input type="submit" value="Submit">
    <input type="reset" value="Reset Form">
</p>
</form>
</body>
</html>
```

### Code Explanation

This code starts by defining regular expressions for an email address and a password. Let's break each one down.

```
var RE_EMAIL = /^(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+$;/
```

1. The caret (^) says to start at the beginning. This prevents the user from entering invalid characters at the beginning of the email address.
2. (\\w+[-\\.])\* allows for a sequence of word characters followed by a dot or a dash. The \* indicates that the pattern can be repeated zero or more times. Successful patterns include "ndunn.", "ndunn-", "nat.s.", and "nat-s-".
3. \\w+ allows for one or more word characters.

## JAVA SCRIPT

---

4. @ allows for a single @ symbol.
5. (w+\.)+ allows for a sequence of word characters followed by a dot. The + indicates that the pattern can be repeated one or more times. This is the domain name without the last portion (e.g, without the "com" or "gov").
6. [A-Za-z]+ allows for one or more letters. This is the "com" or "gov" portion of the email address.
7. The dollar sign (\$) says to end here. This prevents the user from entering invalid characters at the end of the email address.

```
var RE_PASSWORD = /^[A-Za-z\d]{6,8}$/;
```

1. The caret (^) says to start at the beginning. This prevents the user from entering invalid characters at the beginning of the password.
2. [A-Za-z\d]{6,8} allows for a six- to eight-character sequence of letters and digits.
3. The dollar sign (\$) says to end here. This prevents the user from entering invalid characters at the end of the password.

### ***Exercise: Advanced Form Validation***

Duration: 25 to 40 minutes.

1. Open [RegularExpressions/Exercises/FormValidation.js](#) for editing.
    - o Write additional regular expressions to check for:
      1. Proper Name
        - starts with capital letter
        - followed by one or more letters or apostrophes
        - may be multiple words (e.g, "New York City")
      2. Initial
        - zero or one capital letters
      3. State
        - two capital letters
      4. US Postal Code
        - five digits (e.g, "02138")
        - possibly followed by a dash and four digits (e.g, "-1234")
      5. Username
        - between 6 and 15 letters or digits
  2. Open [RegularExpressions/Exercises/Register.html](#) for editing.
    - o Add validation to check the following fields:
      1. first name
      2. middle initial
      3. last name
      4. city
      5. state
      6. zip
      7. username
  3. Test your solution in a browser.
    1. Add regular expressions to test Canadian and United Kingdom postal codes:
      - o Canadian Postal Code - A letter followed by a digit, a letter, a space, a digit, a letter, and a digit (e.g, M1A 1A1)
      - o United Kingdom Postal Code - One or two letters followed by a digit, an optional letter, a space, a digit, and two letters (e.g, WC1N 3XX)
    2. Modify [Register.html](#) to check the postal code against these two new regular expressions as well as the regular expression for a US postal code.
-

[Where is the solution?](#)

## Cleaning Up Form Entries

It is sometimes nice to clean up user entries immediately after they are entered. This can be done using a combination of regular expressions and the `replace()` method of a string object.

### The `replace()` Method Revisited

[Earlier](#), we showed how the `replace()` method of a string object can be used to replace regular expression matches with a string. The `replace()` method can also be used with backreferences to replace a matched pattern with a new string made up of substrings from the pattern. The example below illustrates this.

### Code Sample: *RegularExpressions/Demos/SsnCleaner.html*

```
<html>
<head>
<title>ssn Cleaner</title>
<script type="text/javascript">
var RE_SSN = /^(\d{3})[\- ]?(\d{2})[\- ]?(\d{4})$/;

function cleanSsn(ssn) {
  if (RE_SSN.test(ssn)) {
    var cleanedSsn = ssn.replace(RE_SSN, "$1-$2-$3");
    return cleanedSsn;
  } else {
    alert("INVALID SSN");
    return ssn;
  }
}
</script>
</head>
<body>
  <form onsubmit="return false;">
    <input type="text" name="ssn" size="20">
    <input type="button" value="Clean SSN"
      onclick="this.form.ssn.value = cleanSsn(this.form.ssn.value);">
  </form>
</body>
</html>
```

#### Code Explanation

The `cleanSsn()` function is used to "clean up" a social security number. The regular expression contained in `RE_SSN`, `^(\d{3})[\- ]?(\d{2})[\- ]?(\d{4})$`, contains three subexpressions: `(\d{3})`, `(\d{2})`, and `(\d{4})`. Within the `replace()` method, these subexpressions can be referenced as `$1`, `$2`, and `$3`, respectively.

When the user clicks on the "Clean SSN" button, the `cleanSsn()` function is called. This function first tests to see that the user-entered value is a valid social security number. If it is, it then cleans it up with the line of code below, which dash-delimits the three substrings matching the subexpressions.

```
var cleanedSsn = ssn.replace(RE_SSN, "$1-$2-$3");
```

## JAVA SCRIPT

---

It then returns the cleaned-up social security number.

### ***Exercise: Cleaning Up Form Entries***

Duration: 15 to 25 minutes.

1. Open [RegularExpressions/Exercises/PhoneCleaner.html](#) for editing.
2. Where the comment indicates, declare a variable called `cleanedPhone` and assign it a cleaned-up version of the user-entered phone number. The cleaned up version should fit the following format:  
(555) 555-1212
3. Test your solution in a browser.

Some phone numbers are given as a combination of numbers and letters (e.g, 877-WEBUCATE). As is the case with 877-WEBUCATE, such numbers often have an extra character just to make the word complete.

1. Add a function called `convertPhone()` that:
  - strips all characters that are not numbers or letters
  - converts all letters to numbers
    - ABC -> 2
    - DEF -> 3
    - GHI -> 4
    - JKL -> 5
    - MNO -> 6
    - PRS -> 7
    - TUV -> 8
    - WXY -> 9
    - QZ -> 0
  - passes the first 10 characters of the resulting string to the `cleanPhone()` function
  - returns the resulting string
2. Modify the form, so that it calls `convertPhone()` rather than `cleanPhone()`.
3. Test your solution in a browser.

[Where is the solution?](#)

## **Regular Expressions Conclusion**

In this lesson of the JavaScript tutorial, you have learned to work with regular expressions to validate and to clean up form entries.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## Dynamic Forms

In this lesson of the JavaScript tutorial, you will learn...

1. To create jump menus.
2. To create interdependent select menus.
3. To create a JavaScript TIMER
4. To build a simple JavaScript testing tool.

## Jump Menu

A *jump menu* is a select menu that contains a list of websites or pages to visit. There are two main types of jump menu. One jumps to the selected page as soon as the user makes a selection. The other jumps to a page only after the user has made a selection and clicked on a "Go" button. We'll look at the latter type first and then create the former type in an exercise.

### ***Code Sample: DynamicForms/Demos/JumpMenus.html***

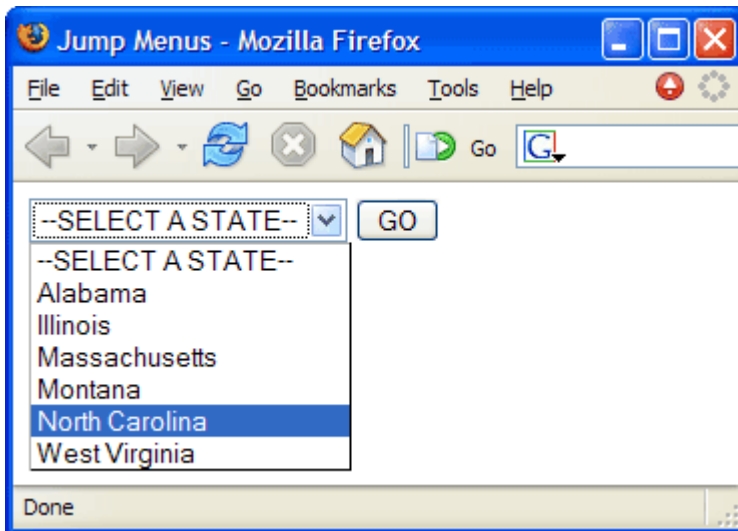
```
<html>
<head>
<title>Jump Menu</title>
<script type="text/javascript">
function jumpMenu(select) {
```

## JAVA SCRIPT

```
var i = select.selectedIndex;
var selection = select.options[i].value;
var url;
if (i === 0) {
    alert("Please select a state.");
} else {
    url = "http://www.50states.com/" + selection + ".htm";
    location.href = url;
}
}
</script>
</head>
<body>
<form>
<select name="State">
<option value="0">--SELECT A STATE--</option>
<option value="alabama">Alabama</option>
<option value="illinois">Illinois</option>
<option value="massachu">Massachusetts</option>
<option value="montana">Montana</option>
<option value="ncarolin">North Carolina</option>
<option value="wvirgini">West Virginia</option>
</select>
<input type="button" value="GO"
    onclick="jumpMenu(this.form.State);">
</form>
</body>
</html>
```

Code Explanation

Viewed in a browser, the page looks like this:



### *The options[] Array*

Select menus contain an array of options. Like with all JavaScript arrays, the first item has an index of 0. So, in this example, the first option, which reads "--SELECT A STATE--" is option 0.

### *The selectedIndex Property*

Select menus have a selectedIndex property that holds the index of the currently selected option. For example, in the sample above, if North Carolina is selected, State.selectedIndex would hold 5. To see if the first option of a select menu is selected, check if selectedIndex is 0.

Let's look at the code above in detail.

1. The "GO" button has an onclick event handler that, when clicked, passes the State select menu to the jumpMenu() function.

```
<input type="button" value="GO" onclick="jumpMenu(this.form.State);">
```

2. The jumpMenu() function does three things:
  1. Creates the variable i that holds the selectedIndex property of the passed-in select menu.
  2. Creates the variable selection that holds the value of the selected option.
  3. Checks to see if the first option is selected.
    - If it is, the function alerts the user to select an option.
    - If it is not, the function creates a variable, url, to hold the destination page and then loads that page by changing the href property of the location object to url.

### *Disabling Form Elements*

Form elements can be disabled by setting the element's disabled property to true. They can be re-enabled by setting the disabled property to false.

#### Syntax

```
FormElement.disabled = true;  
FormElement.disabled = false;
```

The example below is a modified version of the jump menu that disables the "GO" button unless a state is selected.

### *Code Sample: DynamicForms/Demos/JumpMenus2.html*

```
<html>  
<head>  
<title>Jump Menu</title>  
<script type="text/javascript">  
function jumpMenu(select) {  
  var i = select.selectedIndex;  
  var selection = select.options[i].value;  
  var url = "http://www.50states.com/" + selection + ".htm";  
  location.href = url;  
}  
  
function toggleButton(form) {  
  if (form.State.selectedIndex === 0) {  
    form.btnJump.disabled = true;  
  } else {  
    form.btnJump.disabled = false;  
  }  
}  
</script>
```

## JAVA SCRIPT

```
</head>
<body onload="toggleButton (document.forms [0] );">
<form>
  <select name="State" onChange="toggleButton (this.form) ;">
    <option value="0">--SELECT A STATE--</option>
    <option value="alabama">Alabama</option>
    <option value="illinois">Illinois</option>
    <option value="massachu">Massachusetts</option>
    <option value="montana">Montana</option>
    <option value="ncarolin">North Carolina</option>
    <option value="wvirgini">West Virginia</option>
  </select>
  <input type="button" name="btnJump" value="GO"
    onclick="jumpMenu (this.form.State) ;">
</form>
</body>
</html>
```

### Code Explanation

Notice that the jumpMenu() function no longer checks if a state has been selected. It is only called when the user clicks the "Go" button, which is only enabled if a state is selected.

### ***Exercise: Modifying the Jump Menu***

Duration: 15 to 25 minutes.

In this exercise, you will modify the jump menu from the demo so that it jumps to the selected page as soon as the user makes a selection.

1. Open [DynamicForms/Exercises/JumpMenus.html](#) for editing.
2. Add an onchange event handler to the select menu that calls jumpMenu() and passes in itself.
3. Modify the function so that it no longer alerts the user when the first option is selected.
4. Remove the "GO" button from the form.
5. Test the solution in a browser.

The jumpMenu() function isn't reusable. Why not? Fix the function so that it is portable and then modify the form accordingly.

[Where is the solution?](#)

## Interdependent Select Menus

Interdependent select menus allow you to populate one select menu based on a choice made in another select menu. For example, a menu of cities might change based on the state that was selected in another menu, as shown in the following example.

### ***Code Sample: DynamicForms/Demos/Interdependent.html***

```
<html>
<head>
<title>Interdependent Select Menus</title>
<script type="text/javascript">
var NEW_YORKERS = [];
var CALIFORNIANS = [];
```



## JAVA SCRIPT

```
NEW_YORKERS[0] = new Option("New York City", "NYC");
NEW_YORKERS[1] = new Option("Syracuse", "SYR");
NEW_YORKERS[2] = new Option("Albany", "ALB");
NEW_YORKERS[3] = new Option("Rochester", "ROC");

CALIFORNIANS[0] = new Option("Los Angeles", "LAN");
CALIFORNIANS[1] = new Option("San Diego", "SDI");
CALIFORNIANS[2] = new Option("San Francisco", "SFR");
CALIFORNIANS[3] = new Option("Oakland", "OAK");

function populateSub(mainSel, subSel){
    var mainMenu = mainSel;
    var subMenu = subSel;
    var subMenuItems;
    subMenu.options.length = 0;

    switch (mainMenu.selectedIndex) {
        case 0:
            subMenuItems = NEW_YORKERS;
            break;
        case 1:
            subMenuItems = CALIFORNIANS;
            break;
    }

    for (var i = 0; i < subMenuItems.length; i++) {
        subMenu.options[i] = subMenuItems[i];
    }
}

</script>
</head>
<body>
<form name="Menus">
    <select name="State" onchange="populateSub(this, this.form.City);">
        <option value="NY">New York</option>
        <option value="CA">California</option>
    </select>
    <select name="City">
        <option value="NYC">New York City</option>
        <option value="SYR">Syracuse</option>
        <option value="ALB">Albany</option>
        <option value="ROC">Rochester</option>
    </select>
</form>
</body>
</html>
```

### Code Explanation

Let's look at the code above in detail.

1. As the page loads, two arrays (NEW\_YORKERS and CALIFORNIANS) are created and then populated with Options using the Option() constructor. The Option() constructor takes four parameters: text, value, defaultSelected, and selected. Only text is required.
2. The body of the page contains a form with two select menus: State and City. When a state is selected, the onchange event handler triggers a call to the populateSub() function, passing in the State menu as mainSel and the City menu as subSel.

## JAVA SCRIPT

3. A few variables are created and then the City menu is emptied with the following code.

```
subMenu.options.length = 0;
```

4. A switch statement based on the option selected in the main menu is used to determine the array to look in for the submenu's options.

```
5.  switch (mainMenu.selectedIndex) {
6.  case 0:
7.    arrSubMenu = NEW_YORKERS;
8.    break;
9.  case 1:
10.   arrSubMenu = CALIFORNIANS;
11.   break;
    }
```

12. A for loop is used to loop through the array populating the submenu with options.

### ***Making the Code Modular***

A problem with the code in `DynamicForms/Demos/Interdependent.html` is that it would need to be modified in several places to add or change options. This next example, though more complex, is much more modular, and hence, reusable.

### ***Code Sample: DynamicForms/Demos/Interdependent2.html***

```
<html>
<head>
<title>Interdependent Select Menus</title>
<script type="text/javascript">
var MENU = [];
MENU[0] = [];
MENU[1] = [];

MENU[0][0] = new Option("New York", "NY");
MENU[0][1] = new Option("New York City", "NYC");
MENU[0][2] = new Option("Syracuse", "SYR");
MENU[0][3] = new Option("Albany", "ALB");
MENU[0][4] = new Option("Rochester", "ROC");

MENU[1][0] = new Option("California", "CA");
MENU[1][1] = new Option("Los Angeles", "LAN");
MENU[1][2] = new Option("San Diego", "SDI");
MENU[1][3] = new Option("San Francisco", "SFR");
MENU[1][4] = new Option("Oakland", "OAK");

function populateMain(mainSel, subSel){
  var mainMenu = mainSel;
  var subMenu = subSel;
  mainMenu.options.length = 0;
  for (var i = 0; i < MENU.length; i++) {
    mainMenu.options[i] = MENU[i][0];
  }
  populateSub(mainMenu, subMenu);
}

function populateSub(mainSel, subSel){
  var mainMenu = mainSel;
```

## JAVA SCRIPT

```
var subMenu = subSel;
var optMainMenu;
subMenu.options.length = 1;
optMainMenu = mainMenu.selectedIndex;
for (var i = 1; i < MENU[optMainMenu].length; i++) {
    subMenu.options[i] = MENU[optMainMenu][i];
}
}
</script>
</head>
<body onload="populateMain(document.Menus.State, document.Menus.City);">
<form name="Menus">
    <select name="State" onchange="populateSub(this, this.form.City);"></select>
    <select name="City">
        <option value="0">--Please Choose--</option>
    </select>
</form>
</body>
</html>
```

### Code Explanation

This example uses a two-dimensional array to hold the menus. The first item of each array holds the State options, which is used in the main menu. The rest of the items in each array hold the City options used to populate the submenu.

The State select menu starts out empty and the City menu starts out with just a single "Please Choose" option. The two functions `populateMain()` and `populateSub()` are used to populate the two menus. Both functions are completely generic and reusable.

### ***Exercise: Creating Interdependent Select Menus***

Duration: 20 to 30 minutes.

In this exercise, you will modify [DynamicForms/Exercises/Interdependent.html](#), so that the second select menu is dependent on the first.

1. Open [DynamicForms/Exercises/Interdependent.html](#) for editing.
2. Notice that an external JavaScript file, [Select.js](#), is included. This file is shown below.
3. An array, `MENU`, is created and populated with four internal arrays: `MENU[0]`, `MENU[1]`, `MENU[2]`, and `MENU[3]`.
4. Populate the arrays so that:
  - The Bands select menu will be populated with "Beatles", "Rolling Stones", "Genesis", and "Eagles".
  - When "Beatles" is chosen from the Bands select menu, the Artists select menu contains:
    - Paul McCartney with the value of "http://www.paulmccartney.com"
    - John Lennon with the value of "http://www.johnlennon.it"
    - George Harrison with the value of "http://www.georgeharrison.com"
    - Ringo Starr with the value of "http://www.ringostarr.com"
  - When "Rolling Stones" is chosen, the Artists select menu contains:
    - Mick Jagger with the value of "http://www.mickjagger.com"
    - Keith Richards with the value of "http://www.keithrichards.com"
    - Charlie Watts with the value of "http://www.rosebudus.com/watts"
    - Bill Wyman with the value of "http://www.billwyman.com"
  - When "Genesis" is chosen, the Artists select menu contains:

## JAVA SCRIPT

- Phil Collins with the value of "http://www.philcollins.co.uk"
  - Peter Gabriel with the value of "http://www.petergabriel.com"
  - Mike Rutherford with the value of "http://www.mikemechanics.com"
  - When "Eagles" is chosen, the Artists select menu contains:
    - Don Henley with the value of "http://www.donhenley.com"
    - Joe Walsh with the value of "http://www.joewalsh.com"
    - Glenn Frey with the value of "http://www.imdb.com/name/nm0004940"
5. Change the values of arg1 and arg2 in the calls to populateMain() and populateSub() in the event handlers in the HTML code so that the correct arguments are passed to these functions.

### ***Code Sample: DynamicForms/Exercises/Select.js***

```
function populateMain(mainSel, subSel) {
    var mainMenu = mainSel;
    var subMenu = subSel;
    mainMenu.options.length = 0;
    for (var i = 0; i < MENU.length; i++) {
        mainMenu.options[i] = MENU[i][0];
    }
    populateSub(mainMenu, subMenu);
}

function populateSub(mainSel, subSel) {
    var mainMenu = mainSel;
    var subMenu = subSel;
    var optMainMenu;
    subMenu.options.length = 1;
    optMainMenu = mainMenu.selectedIndex;
    for (var i = 1; i < MENU[optMainMenu].length; i++) {
        subMenu.options[i] = MENU[optMainMenu][i];
    }
}

function jumpMenu(select) {
    var i = select.selectedIndex;
    var url = select.options[i].value;
    if (i > 0) {
        location.href = url;
    }
}
```

### ***Code Sample: DynamicForms/Exercises/Interdependent.html***

```
<html>
<head>
<title>Interdependent Select Menus</title>
<script type="text/javascript" src="Select.js"></script>
<script type="text/javascript">
var MENU = [];
MENU[0] = [];
MENU[1] = [];
MENU[2] = [];
MENU[3] = [];

/*
Populate the arrays so that:
-The Bands select menu will be populated with
  Beatles, Rolling Stones, Genesis, and Eagles
```

## JAVA SCRIPT

```
-When Beatles is chosen, the Artists select menu contains:
TEXT      VALUE
Paul McCartney http://www.paulmccartney.com
John Lennon  http://www.johnlennon.it
George Harrison http://www.georgeharrison.com
Ringo Starr  http://www.ringostarr.com

-When Rolling Stones is chosen, the Artists select menu contains:
TEXT      VALUE
Mick Jagger  http://www.mickjagger.com
Keith Richards http://www.keithrichards.com
Charlie Watts http://www.rosebudus.com/watts
Bill Wyman   http://www.billwyman.com

-When Genesis is chosen, the Artists select menu contains:
TEXT      VALUE
Phil Collins http://www.philcollins.co.uk
Peter Gabriel http://www.petergabriel.com
Mike Rutherford http://www.mikemechanics.com

-When Eagles is chosen, the Artists select menu contains:
TEXT      VALUE
Don Henley   http://www.donhenley.com
Joe Walsh    http://www.joewalsh.com
Glenn Frey   http://www.imdb.com/name/nm0004940

Change the values of arg1 and arg2 in the calls to populateMain()
and populateSub() in the event handlers in the HTML code
so that the correct arguments are passed to these functions.
*/

</script>
</head>
<body onload="populateMain(arg1, arg2);">
<form name="Menus">
  Band: <select name="Bands" onchange="populateSub(arg1, arg2);"></select>
  Artist: <select name="Artists" onchange="jumpMenu(this);">
    <option value="0">--Please Choose--</option>
  </select>
</form>
</body>
</html>
```

[Where is the solution?](#)

## Creating a JavaScript Timer

JavaScript timers can be used to create timed quizzes or events. The trick to a timer is to call a function recursively on a time increment.

### The setTimeout() Method

The window object's setTimeout() method is used to execute a block of JavaScript code every n milliseconds. The syntax is shown below.

```
setTimeout("statements to execute", n);
```

## JAVA SCRIPT

---

Or

```
setTimeout(functionToCall, n);
```

The example below shows how the `setTimeout()` method can be used to create a timer.

### ***Code Sample: DynamicForms/Demos/Timer.html***

```
<html>
<head>
<title>JavaScript Timer</title>
<script type="text/javascript">

var SECONDS_LEFT, TIMER, TIMES_UP;
function init(){
  document.Timer.btnStart.disabled = true;
}

function resetTimer(seconds){
  SECONDS_LEFT = seconds;
  document.Timer.TimeLeft.value = SECONDS_LEFT;
  clearTimeout(TIMER);
  document.Timer.btnStart.disabled = false;
  document.Timer.btnReset.disabled = true;
}

function decrementTimer(){
  TIMES_UP = false;
  document.Timer.TimeLeft.value = SECONDS_LEFT;
  document.Timer.btnStart.disabled = true;
  document.Timer.btnReset.disabled = false;
  SECONDS_LEFT--;
  if (SECONDS_LEFT >= 0) {
    TIMER = setTimeout(decrementTimer, 1000);
  } else {
    alert("Time's up!");
    resetTimer(10);
  }
}
</script>
</head>
<body onload="init();">
<form name="Timer" onsubmit="return false;">
Timer: <input type="text" name="TimeLeft" size="2"
  style="text-align:center" onfocus="this.blur();">
  seconds left<br>
  <input type="button" name="btnStart"
    value="Start" onclick="decrementTimer();">
  <input type="button" name="btnReset"
    value="Reset" onclick="resetTimer(10);">
</form>
</body>
</html>
```

Code Explanation

Let's look at the code above in detail.

## JAVA SCRIPT

- 
1. As the page loads, three global variables are created: SECONDS\_LEFT to hold the number of seconds left, TIMER - to hold the timer, and TIMES\_UP - to flag if the timer has run out.

```
var SECONDS_LEFT, TIMER, TIMES_UP;
```

2. The body of the page contains a form with a text field holding the number of seconds left, a "Start" button and a "Reset" button.

```
3. <form name="Timer" onsubmit="return false;">
4.   Timer: <input type="text" name="TimeLeft" size="2"
5.         style="text-align:center" onfocus="this.blur();">
6.         seconds left<br>
7.         <input type="button" name="btnStart"
8.               value="Start" onclick="decrementTimer();">
9.         <input type="button" name="btnReset"
10.        value="Reset" onclick="resetTimer(10);">
    </form>
```

11. The onload event handler of the body tag triggers the init() function, which disables the "Start" button.

```
12. function init(){
13.   document.Timer.btnStart.disabled = true;
    }
```

14. When the user clicks on the "Reset" button, the resetTimer() function is called. This function does the following:

- o Sets SECONDS\_LEFT to the number passed into the function.

```
SECONDS_LEFT = seconds;
```

- o Sets the value of the TimeLeft text field to SECONDS\_LEFT.

```
document.Timer.TimeLeft.value = SECONDS_LEFT;
```

- o Clears the TIMER timer.

```
clearTimeout(TIMER);
```

- o Enables the "Start" button.

```
document.Timer.btnStart.disabled = false;
```

- o Disables the "Reset" button.

```
document.Timer.btnReset.disabled = true;
```

15. When the user clicks on the "Start" button, the decrementTimer() function is called. This function does the following:

- o Sets TIMES\_UP to false.

```
TIMES_UP = false;
```

- o Sets the value of the TimeLeft text field to SECONDS\_LEFT.

## JAVA SCRIPT

---

```
document.Timer.TimeLeft.value = SECONDS_LEFT;
```

- Disables the "Start" button.

```
document.Timer.btnStart.disabled = true;
```

- Enables the "Reset" button.

```
document.Timer.btnReset.disabled = false;
```

- Decrements SECONDS\_LEFT by one.

```
SECONDS_LEFT--;
```

- Checks to see if SECONDS\_LEFT is greater than or equal to zero.

```
○   if (SECONDS_LEFT >= 0) {  
○     TIMER = setTimeout(decrementTimer, 1000);  
○   } else {  
○     alert("Times up!");  
○     resetTimer(10);  
○   }
```

- If SECONDS\_LEFT is greater than or equal to zero, setTimeout() is used to re-call decrementTimer() after 1000 milliseconds (1 second). This creates a timer object, which is assigned to TIMER.
- If SECONDS\_LEFT is less than zero, an alert pops up notifying the user that time is up and resetTimer(), which clears the timer, is called.

### ***Exercise: Adding a "Pause" Button to the Timer***

Duration: 10 to 20 minutes.

In this exercise, you will add a "Pause" button to the timer.

1. Open [DynamicForms/Exercises/Timer.html](#) for editing.
2. This page is the same as [DynamicForms/Demos/Timer.html](#), except that it has a new "Pause" button that, when clicked, calls the pauseTimer() function. Your job is to create this pauseTimer() function and to modify the script so that the right buttons are enabled at the right times. The

diagram below shows the four different phases.



## Phase 1

Timer:  seconds left

## Phase 2

Timer:  seconds left

## Phase 3

Timer:  seconds left

## Phase 4

Timer:  seconds left

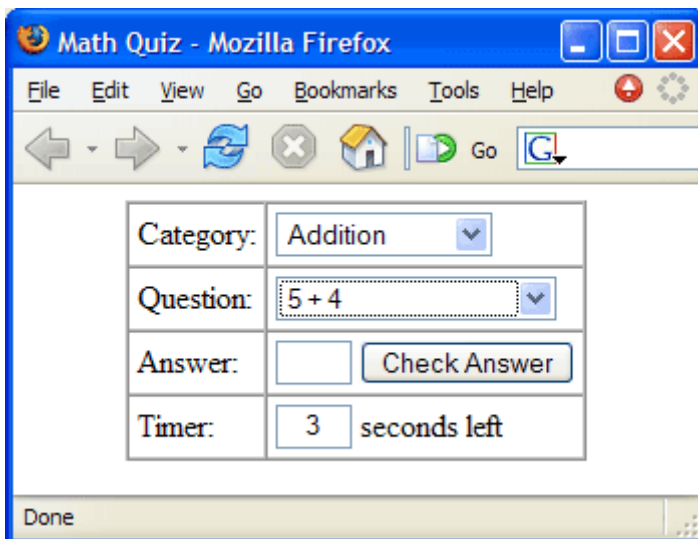
3. Test your solution in a browser.

[Where is the solution?](#)

## A Sample Quiz Tool

The following example brings together the concepts learned in this lesson to create a quiz tool.

The quiz looks like this:



**Code Sample: *DynamicForms/Demos/MathQuiz.html***

```
<html>
```

## JAVA SCRIPT

```
<head>
<title>Math Quiz</title>
<script type="text/javascript" src="Select.js"></script>
<script type="text/javascript">

var MENU = [];
var SECONDS_LEFT, TIMER, TIMES_UP, TOTAL_TIME = 10;

function init(){
    MENU[0] = [];
    MENU[1] = [];
    MENU[2] = [];
    MENU[3] = [];

    MENU[0][0] = new Option("Addition");
    MENU[0][1] = new Option("5 + 4", "9");
    MENU[0][2] = new Option("9 + 3", "12");
    MENU[0][3] = new Option("7 + 12", "19");
    MENU[0][4] = new Option("13 + 24", "37");

    MENU[1][0] = new Option("Subtraction");
    MENU[1][1] = new Option("5 - 1", "4");
    MENU[1][2] = new Option("12 - 4", "8");
    MENU[1][3] = new Option("23 - 11", "12");
    MENU[1][4] = new Option("57 - 19", "38");

    MENU[2][0] = new Option("Multiplication");
    MENU[2][1] = new Option("1 * 3", "3");
    MENU[2][2] = new Option("4 * 8", "32");
    MENU[2][3] = new Option("7 * 12", "84");
    MENU[2][4] = new Option("13 * 17", "221");

    MENU[3][0] = new Option("Division");
    MENU[3][1] = new Option("4 / 1", "4");
    MENU[3][2] = new Option("12 / 3", "4");
    MENU[3][3] = new Option("21 / 7", "3");
    MENU[3][4] = new Option("121 / 11", "11");

    populateMain(document.Quiz.Operator, document.Quiz.Question);
    resetTimer(TOTAL_TIME);
    document.Quiz.btnCheck.disabled = true;
    document.Quiz.Answer.disabled = true;
}

function resetTimer(seconds){
    TIMES_UP = true;
    SECONDS_LEFT = seconds;
    document.Quiz.TimeLeft.value = SECONDS_LEFT;
    clearTimeout(TIMER);
    document.Quiz.Answer.value = "";
}

function decrementTimer(){
    TIMES_UP = false;
    document.Quiz.TimeLeft.value = SECONDS_LEFT;
    SECONDS_LEFT--;
    if (SECONDS_LEFT >= 0) {
        TIMER = setTimeout(decrementTimer, 1000);
    } else {
```

## JAVA SCRIPT

```
    alert("Time's up! The answer is " + getAnswer() + ".");
    resetTimer(TOTAL_TIME);
}
}

function checkAnswer(answer){
var correctAnswer = getAnswer();
if (answer === correctAnswer) {
    alert("Right! The answer is " + correctAnswer + ".");
} else {
    alert("Sorry. The correct answer is " + correctAnswer + ".");
}
removeOption();
questionChange();
}

function removeOption(){
var i = document.Quiz.Operator.selectedIndex;
var j = document.Quiz.Question.selectedIndex;
MENU[i].splice(j, 1);
if (MENU[i].length == 1) {
    MENU.splice(i, 1);
    if (MENU.length === 0) {
        endQuiz();
    }
}
populateMain(document.Quiz.Operator, document.Quiz.Question);
resetTimer(TOTAL_TIME);
}

function questionChange(){
if (document.Quiz.Question.selectedIndex === 0) {
    document.Quiz.btnCheck.disabled = true;
    document.Quiz.Answer.disabled = true;
    resetTimer(TOTAL_TIME);
} else {
    document.Quiz.btnCheck.disabled = false;
    document.Quiz.Answer.disabled = false;
    decrementTimer();
}
}

function endQuiz(){
    resetTimer(TOTAL_TIME);
    alert("Thanks for playing! The quiz will now reload.");
    init();
}

function getAnswer(){
    var i = document.Quiz.Question.selectedIndex;
    var answer = document.Quiz.Question[i].value;
    return answer;
}

</script>
</head>
<body onload="init();">
<form name="Quiz" onsubmit="return false;">
<table border="1" cellspacing="0" cellpadding="4" align="center">
```

## JAVA SCRIPT

```
<tr>
  <td>Category:</td>
  <td>
    <select name="Operator"
      onchange="populateSub(this, this.form.Question);
        resetTimer(TOTAL_TIME);">
    </select>
  </td>
</tr>
<tr>
  <td>Question:</td>
  <td>
    <select name="Question"
      onchange="questionChange();">
      <option value="0">--Please Choose--</option>
    </select>
  </td>
</tr>
<tr>
  <td>Answer:</td>
  <td>
    <input type="text" name="Answer" size="2">
    <input type="button" name="btnCheck" value="Check Answer"
      onclick="checkAnswer(this.form.Answer.value);">
  </td>
</tr>
<tr>
  <td>Timer:</td>
  <td><input type="text" name="TimeLeft" size="2"
    style="text-align:center" onfocus="this.blur();">
    seconds left</td>
</tr>
</table>
</form>
</body>
</html>
```

### Code Explanation

Here's how the quiz works:

1. The Question select menu is always populated with questions in the indicated category.
2. The "Check Answer" button and the Answer text field are only enabled when a question is selected.
3. The timer starts when a question is selected and stops when it runs out or when the "Check Answer" button is clicked.
4. When the "Check Answer" button is clicked, the user is alerted as to whether or not the answer is correct and the question is removed from the question menu.
5. When all questions in a category are gone, the category is removed from the category menu.
6. When all questions in all categories have been completed, the user is thanked for taking the quiz and the entire quiz is restarted.

Spend some time reviewing this code. You shouldn't see anything new, except in the way the code is designed.

## Dynamic Forms Conclusion

In this lesson of the JavaScript tutorial, you have learned to build interactive, dynamic forms.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

## Dynamic HTML

**In this lesson of the JavaScript tutorial, you will learn...**

1. To change the values of CSS properties dynamically.
2. To hide and show elements.
3. To dynamically modify the content fo elements.
4. To manipulate tables dynamically.
5. To position elements dynamically.
6. To change the z-index order of elements.

## Introduction

Dynamic HTML is not a technology in and of itself, but rather is a combination of three technologies: HTML, Cascading Style Sheets (CSS), and JavaScript. It usually involves using JavaScript to change a CSS property of an HTML element. In modern browsers, most CSS

## JAVA SCRIPT

---

properties can be modified dynamically. This can be done by changing an individual style of element (using the style property of the element) or by changing the class name assigned to the element (using the className property).

### Accessing and Modifying Styles

The style object is a collection of an element's styles that are either defined within that HTML element's style attribute or directly in JavaScript. Styles defined in the <style> tag or in an external style sheet are not part of the style object. The W3C specifies a method for getting at the current (or actual) style of an object: the window object's getComputedStyle() method, which is supported by the latest versions of Mozilla, but not by Internet Explorer 6 and earlier. Internet Explorer provides a non-standard property for getting at the current style of an element: the currentStyle property.

#### Standard Syntax

```
window.getComputedStyle(Element, Pseudo-Element)
```

```
//for example:  
var curStyle = window.getComputedStyle(  
    document.getElementById("divTitle"), null);  
alert(curStyle.fontWeight);
```

#### Internet Explorer Syntax

```
Element.currentStyle.Property  
//for example:  
alert(document.getElementById("divTitle").currentStyle.fontWeight);
```

#### Cross-browser Solution

```
var curStyle;  
if (window.getComputedStyle) {  
    curStyle = window.getComputedStyle(  
        document.getElementById("divTitle"), "");  
} else {  
    curStyle = document.getElementById("divTitle").currentStyle;  
}  
alert(curStyle.fontWeight);
```

Another solution is to use JavaScript to set the styles of the objects on the page.

```
function init(){  
    document.getElementById("divTitle").style.fontWeight = "bold";  
}
```

Now this style can be referenced later in the script; however, this solution isn't always practical.

### Hiding and Showing Elements

Elements can be hidden and shown by changing their visibility or display values. The visibility style can be set to visible or hidden and the display property can be set to block, table-row, none, and several other values. The two work slightly differently as the following example illustrates.

## JAVA SCRIPT

### *Code Sample: DynamicHtml/Demos/Visibility.html*

```
<html>
<head>
<title>Showing and Hiding Elements with JavaScript</title>
<script type="text/javascript" src="EnvVars.js"></script>
<script type="text/javascript">

function changeVisibility(TR) {
  if (document.getElementById(TR).style.visibility=="hidden") {
    document.getElementById(TR).style.visibility = "visible";
  } else {
    document.getElementById(TR).style.visibility = "hidden";
  }
}

var TR_DISPLAY = (IS_IE) ? "block" : "table-row";

function changeDisplay(TR) {
  if (document.getElementById(TR).style.display == "none") {
    document.getElementById(TR).style.display = TR_DISPLAY;
  } else {
    document.getElementById(TR).style.display = "none";
  }
}
</script>
</head>
<body>
<h1>Hiding and Showing Elements</h1>
<table border="1">
  <tr id="tr1"><td>tableElem Row 1</td></tr>
  <tr id="tr2"><td>tableElem Row 2</td></tr>
  <tr id="tr3"><td>tableElem Row 3</td></tr>
  <tr id="tr4"><td>tableElem Row 4</td></tr>
</table>
<form>
<h2>visibility</h2>
<input type="button" onclick="changeVisibility('tr1')" value="TR1">
<input type="button" onclick="changeVisibility('tr2')" value="TR2">
<input type="button" onclick="changeVisibility('tr3')" value="TR3">
<input type="button" onclick="changeVisibility('tr4')" value="TR4">

<h2>display</h2>
<input type="button" onclick="changeDisplay('tr1')" value="TR1">
<input type="button" onclick="changeDisplay('tr2')" value="TR2">
<input type="button" onclick="changeDisplay('tr3')" value="TR3">
<input type="button" onclick="changeDisplay('tr4')" value="TR4">
</form>
</body>
</html>
```

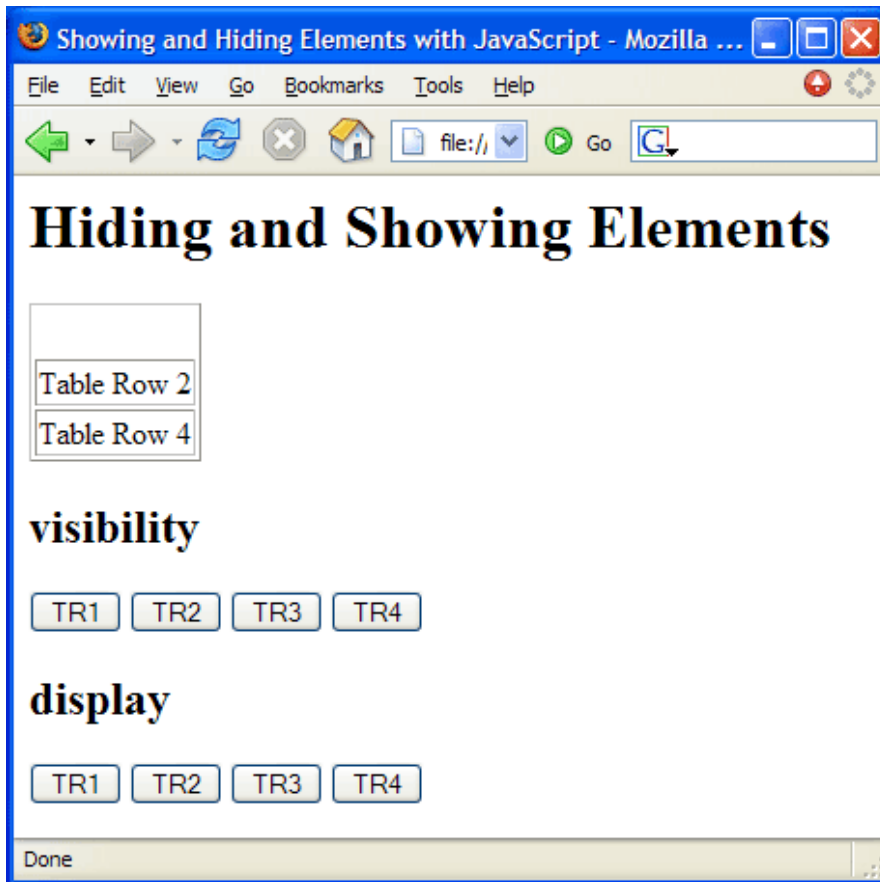
#### Code Explanation

This page has two functions: `changeVisibility()` and `changeDisplay()`. The `changeVisibility()` function checks the value of the visibility style of the passed element and changes it to its opposite. The `changeDisplay()` function does the same with the display style. The functions are called with buttons on the page and are passed in ids of table rows from the table on the page.

## JAVA SCRIPT

The main difference between setting visibility to hidden and setting display to none is that setting visibility to hidden does not modify the layout of the page; it simply hides the element. Setting display to none, on the other hand, collapses the element, so that the surrounding relatively positioned elements re-position themselves.

In the screenshot below, *tableElem Row 1* has visibility set to hidden and *tableElem Row 3* has display set to none.



The following line of code may have grabbed your attention:

```
var TR_DISPLAY = (IS_IE  
                ) ? "block" : "table-row";
```

According to the CSS specification, the proper way to display a table row is by setting the display property to table-row; however Internet Explorer 6 and earlier do not support this value. To get the script to behave correctly in IE6, the display must be set to block; however, that messes up Mozilla, which displays blocks as blocks, not as table rows. The solution is to create a variable, TR\_DISPLAY, whose value depends on the browser being used. That's exactly what the line of code above does.

### ***Exercise: Showing and Hiding Elements***

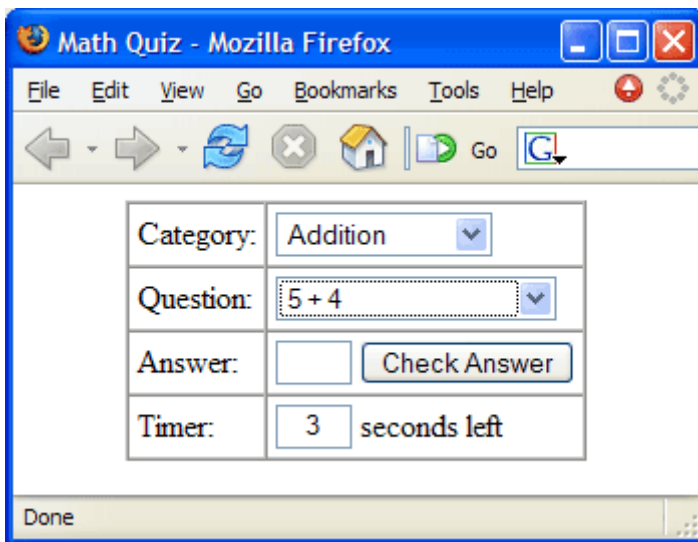
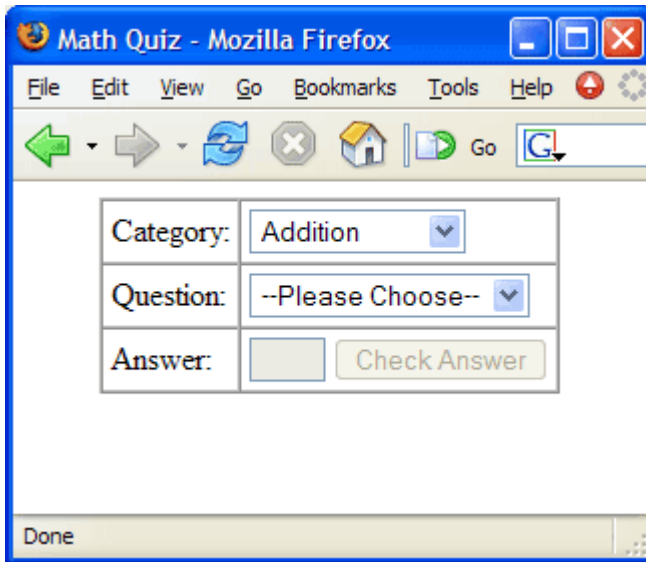
Duration: 20 to 30 minutes.



## JAVA SCRIPT

In this exercise, you will modify a Math Quiz to only show the countdown timer when it is running.

1. Open [DynamicHtml/Exercises/MathQuiz.html](#) for editing.
2. Modify the code so that the table row with the timer in it (the last row) only shows up when the timer is running. Note that the variable TR\_DISPLAY is already set to the proper browser-dependent value.



*Hint:* You will make your changes in `resetTimer()` and `decrementTimer()`.

3. Test your solution in a browser.

Only show the Answer row when a question is selected.

[Where is the solution?](#)

## The innerHTML Property

Internet Explorer 4+ and the latest Mozilla browsers (Netscape 6+ and Firefox) support the non-standard innerHTML property of an element. The innerHTML property can be used to read and set the HTML content of an element. The following example illustrates this.

### Code Sample: *DynamicHtml/Demos/innerHTML.html*

```
<html>
<head>
<title>innerHTML</title>
</head>
<body>
<div id="divGreeting">
  <h1>Hello!</h1>
</div>
<form>
  <input type="button" value="Return Greeting"
    onclick="document.getElementById('divGreetingBack').innerHTML =
      document.getElementById('divGreeting').innerHTML;">
</form>
<div id="divGreetingBack"></div>
</body>
</html>
```

Code Explanation

When the user clicks the "Return Greeting" button, the HTML content of divGreeting is copied into divGreetingBack.

### **Exercise: Using innerHTML for Cleaner Messages**

Duration: 10 to 20 minutes.

In this exercise, you will modify the Math Quiz so that it displays messages in plain text, rather than in form fields and alerts.

1. Open [DynamicHtml/Exercises/MathQuizInnerHTML.html](#) for editing.
2. Replace the TimeLeft text field with a span with an id of "spanTimeLeft".
3. Beneath the form, add an empty div with the id of "divMessage".
4. Create a new function called showMessage(), which takes two arguments: the message to show and the color to display it in. This function should do the following:
  - o Set the innerHTML property of divMessage to the passed-in message.
  - o Change the value of the color style of divMessage to the passed-in color.
  - o Display divMessage as a block.
5. Replace the two alerts in checkAnswer() with calls to showMessage(). Pass in green as the color if the answer is correct and red if it is not.
6. You may find it useful to look at the decrementTimer() function, which already has some modifications.

### Code Sample: *DynamicHtml/Exercises/MathQuizInnerHTML.html*

```
---- Code Omitted ----

function decrementTimer() {
```

## JAVA SCRIPT

```
TIMES_UP = false;
document.getElementById("trTimer").style.display = TR_DISPLAY;

document.getElementById("divMessage").innerHTML = "";
document.getElementById("divMessage").style.display = "none";

document.getElementById("spanTimeLeft").innerHTML=TIME_LEFT + " ";
document.getElementById("trTimer").style.visibility = "visible";

INTERVAL--;
if (TIME_LEFT >= 0) {
    TIMER = setTimeout(decrementTimer, 1000);
} else {
    showMessage("Time's up! The answer is " + getAnswer() + ".", "red");
    resetTimer(INTERVAL);
}
}
---- Code Omitted ----
```

[Where is the solution?](#)

## Manipulating Tables

HTML tables can be created and manipulated dynamically with JavaScript. Each table element contains a rows array and methods for inserting and deleting rows: `insertRow()` and `deleteRow()`. Each tr element contains a cells array and methods for inserting and deleting cells: `insertCell()` and `deleteCell()`. The following example shows how these objects can be used to dynamically create HTML tables.

### *Code Sample: DynamicHtml/Demos/table.html*

```
<html>
<head>
<title>Manipulating Tables</title>
<script type="text/javascript">
function addRow(tableId, cells){
    var tableElem = document.getElementById(tableId);
    var newRow = tableElem.insertRow(tableElem.rows.length);
    var newCell;
    for (var i = 0; i < cells.length; i++) {
        newCell = newRow.insertCell(newRow.cells.length);
        newCell.innerHTML = cells[i];
    }
    return newRow;
}

function deleteRow(tableId, rowNumber){
    var tableElem = document.getElementById(tableId);
    if (rowNumber > 0 && rowNumber < tableElem.rows.length) {
        tableElem.deleteRow(rowNumber);
    } else {
        alert("Failed");
    }
}
</script>
</head>
```

## JAVA SCRIPT

```
<body>
<table id="tblPeople" border="1">
<tr>
  <th>First Name</th>
  <th>Last Name</th>
</tr>
</table>
<hr>
<form name="formName">
  First Name: <input type="text" name="FirstName"><br>
  Last Name: <input type="text" name="LastName"><br>
  <input type="button" value="Add Name"
    onclick="addRow('tblPeople',
      [this.form.FirstName.value, this.form.LastName.value] );">
  <hr>
  Remove Row: <input type="text" size="1" name="RowNum">
  <input type="button" value="Delete Row"
    onclick="deleteRow('tblPeople', this.form.RowNum.value) ">
</form>
</body>
</html>
```

### Code Explanation

The body of the page contains a table with an id of formName. The table contains a single row of headers.

```
<table id="tblPeople" border="1">
<tr>
  <th>First Name</th>
  <th>Last Name</th>
</tr>
</table>
```

Below the table is a form that allows the user to enter a first and last name. When the "Add Name" button is clicked, the addRow() function is called and passed in the id of the table (tblPeople) and a new array containing the user-entered values.

```
First Name: <input type="text" name="FirstName"><br>
Last Name: <input type="text" name="LastName"><br>
<input type="button" value="Add Name"
  onclick="addRow('tblPeople',
    [this.form.FirstName.value, this.form.LastName.value] );">
```

The addRow() function uses the insertRow() method of the table to add a new row at the end of the table and then loops through the passed-in array, creating and populating one cell for each item. The function also returns the new row. Although the returned value isn't used in this example, it can be useful if you then want to manipulate the new row further.

```
function addRow(tableId, cells){
  var tableElem = document.getElementById(tableId);
  var newRow = tableElem.insertRow(tableElem.rows.length);
  var newCell;
  for (var i = 0; i < cells.length; i++) {
    newCell = newRow.insertCell(newRow.cells.length);
    newCell.innerHTML = cells[i];
  }
}
```

## JAVA SCRIPT

---

```
return newRow;
}
```

The form also contains a "Delete Row" button that, when clicked, passes the id of the table (tblPeople) and the number entered by the user in the RowNum text field.

```
Remove Row: <input type="text" size="1" name="RowNum">
<input type="button" value="Delete Row"
onclick="deleteRow('tblPeople', this.form.RowNum.value)">
```

The deleteRow() function checks to see if the row specified exists and is not the first row (the header row). If both conditions are true, it deletes the row. Otherwise, it alerts "Failed".

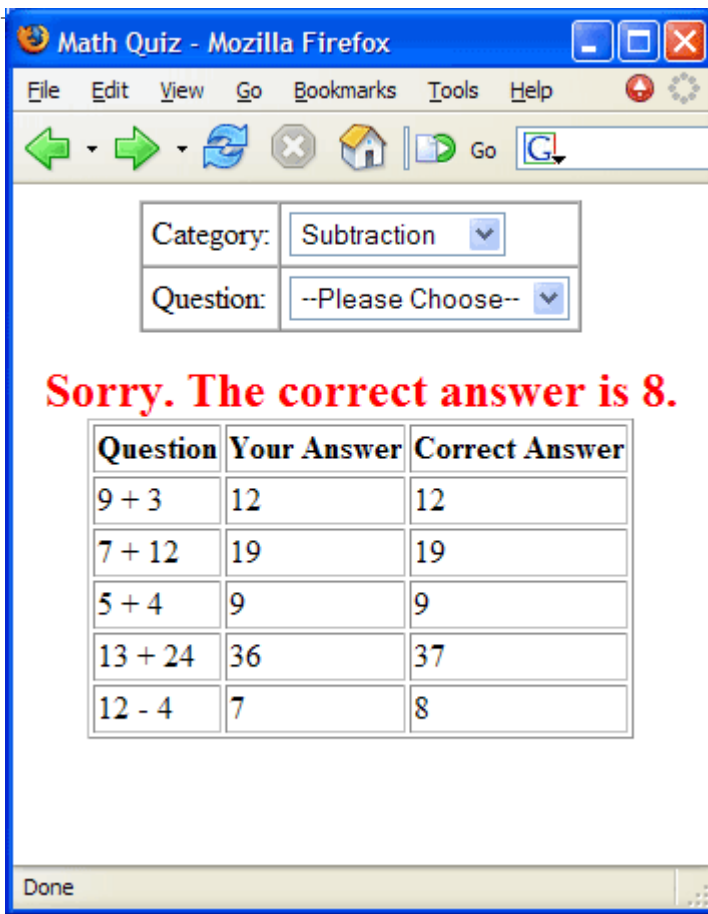
```
function deleteRow(tableId, rowNumber){
var tableElem = document.getElementById(tableId);
if (rowNumber > 0 && rowNumber < tableElem.rows.length) {
tableElem.deleteRow(rowNumber);
} else {
alert("Failed");
}
}
```

### ***Exercise: Tracking Results in the Math Quiz***

Duration: 15 to 25 minutes.

In this exercise, you will dynamically create a table that shows the user how she is doing on the Math Quiz. The screenshot below shows how the result will look.

## JAVA SCRIPT



1. Open [DynamicHtml/Exercises/MathQuizTable.html](#) for editing.
2. Notice that the `addRow()` function is included and that there is an additional function called `getQuestion()`, which returns the question that is currently selected.
3. Also notice that there is a table at the bottom of the body of the page with an id of `tblResults`.
4. Modify the `checkAnswer()` function so that it does the following:
  - o Creates an array to hold the current question and answer.
  - o Adds a new row to the `tblResults` table.
5. Test your solution in a browser.

Modify the code so that the new row's font color is green if the answer is correct and red if it is not. *Hint:* the `addRow()` function returns the newly added row.

[Where is the solution?](#)

## Dynamically Changing Dimensions

The dimensions of an object can be changed by modifying the width and height properties of the element's style property. The following example demonstrates this.

### *Code Sample: DynamicHtml/Demos/Dimensions.html*

```
<html>
<head>
<title>Dimensions</title>
<style type="text/css">
```

## JAVA SCRIPT

```
#divBlock {
  height:100px;
  width:100px;
  background-color:red;
}
</style>
<script type="text/javascript">
function init(){
  document.getElementById("divBlock").style.width = "100px";
  document.getElementById("divBlock").style.height = "100px";
}

function grow(elem){
  var objElem = elem;
  var curWidth = parseInt(objElem.style.width);
  var curHeight = parseInt(objElem.style.height);
  objElem.style.width = (curWidth * 1.5) + 'px';
  objElem.style.height = (curHeight * 1.5) + 'px';
}

function shrink(elem){
  var objElem = elem;
  var curWidth = parseInt(objElem.style.width);
  var curHeight = parseInt(objElem.style.height);
  objElem.style.width = (curWidth / 1.5) + 'px';
  objElem.style.height = (curHeight / 1.5) + 'px';
}
</script>
</head>
<body onload="init();" >
<div id="divBlock" onmouseover="grow(this);"
  onmouseout="shrink(this);"></div>
</body>
</html>
```

### Code Explanation

In this case, we use the `init()` function to set the height and width of the `divBlock` div, thus making the properties accessible to JavaScript.

The `grow()` function uses `parseInt()` to cut off the units (e.g, px) from the value of the width and height of the div and assign the resulting integers to variables: `curWidth` and `curHeight`. It then modifies the width and height properties of the element by multiplying the current values by 1.5.

The `shrink()` function does the same thing, but it divides by 1.5 instead of multiplying.

The functions are triggered with `onmouseover` and `onmouseout` event handlers.

### *Creating a Timed Slider*

The example below shows how a timed slider can be created by dynamically changing an element's dimensions.

### *Code Sample: DynamicHtml/Demos/Slider.html*

```
<html>
<head>
```

## JAVA SCRIPT

```
<title>Slider</title>
<style type="text/css">
#divSlider {
  height:10px;
  width:0px;
  background-color:red;
  position:relative;
  top:-11px;
  left:1px;
}

#divSliderBG {
  height:12px;
  width:102px;
  background-color:blue;
}
</style>
<script type="text/javascript">

var TIMER, TIMES_UP, Slider;
function resetTimer(){
  TIMES_UP = true;
  var slider = document.getElementById("divSlider");
  slider.style.width = "0px";
  clearTimeout(TIMER);
}

function decrementTimer(){
  TIMES_UP = false;
  var slider = document.getElementById("divSlider");
  var curWidth = parseInt(slider.style.width);
  if (curWidth < 100) {
    slider.style.width = curWidth + 1 + "px";
    TIMER = setTimeout(decrementTimer, 100);
  } else {
    alert("Time's up!");
    resetTimer();
  }
}
</script>
</head>

<body onload="resetTimer();">

<div id="divSliderBG"></div>
<div id="divSlider"></div>

<form>
  <input type="button" value="Start Timer" onclick="decrementTimer();">
</form>

</body>
</html>
```



### Positioning Elements Dynamically

The position of an object can be changed by modifying the left and top properties of the element's style property. The following example demonstrates this.

#### *Code Sample: DynamicHtml/Demos/Position.html*

```
<html>
<head>
<title>Position</title>
<style type="text/css">
#divBlock {
  position:relative;
  height:100px;
  width:100px;
  top:100px;
  left:100px;
  background-color:red;
}
</style>
<script type="text/javascript">
function init(){
  document.getElementById("divBlock").style.top = "100px";
  document.getElementById("divBlock").style.left = "100px";
}

function moveH(elem, distance){
  var objElem = document.getElementById(elem);
  var curLeft = parseInt(objElem.style.left);
  objElem.style.left = (curLeft + distance) + "px";
}

function moveV(elem, distance){
  var objElem = document.getElementById(elem);
  var curTop = parseInt(objElem.style.top);
  objElem.style.top = (curTop + distance) + "px";
}
</script>
</head>
<body onload="init();">
<form>
<input type="button" value="Left" onclick="moveH('divBlock',-10);">
<input type="button" value="Right" onclick="moveH('divBlock',10);">
<input type="button" value="Up" onclick="moveV('divBlock',-10);">
<input type="button" value="Down" onclick="moveV('divBlock',10);">
</form>
<div id="divBlock"></div>
</body>
</html>
```

#### Code Explanation

We again use the `init()` function, this time to set the top and left properties of the `divBlock` div, thus making the properties accessible to JavaScript.

## JAVA SCRIPT

The `moveH()` function uses `parseInt()` to cut off the units (e.g, px) from the value of the left property of the div and assign the resulting integer to the `curLeft` variable. It then modifies the left property of the element by adding the value passed in for distance.

The `moveV()` function does the same thing, but it modifies the top property rather than the left property.

The functions are triggered with onclick event handlers.

### *Creating a Different Timed Slider*

The example below shows how a different type of timed slider can be created by dynamically changing an element's position.

#### *Code Sample: DynamicHtml/Demos/Slider2.html*

```
<html>
<head>
<title>Slider</title>
<style type="text/css">
#divSlider {
  height:10px;
  width:2px;
  background-color:red;
  position:relative;
  top:-11px;
  left:1px;
}

#divSliderBG {
  height:12px;
  width:102px;
  background-color:blue;
}
</style>
<script type="text/javascript">

var TIMER, TIMES_UP, Slider;
function resetTimer(){
  TIMES_UP = true;
  var slider = document.getElementById("divSlider");
  slider.style.left = "1px";
  clearTimeout(TIMER);
}

function decrementTimer(){
  TIMES_UP = false;
  var slider = document.getElementById("divSlider");
  var curLeft = parseInt(slider.style.left);
  if (curLeft < 100) {
    slider.style.left = curLeft + 1 + "px";
    TIMER = setTimeout(decrementTimer, 100);
  } else {
    alert("Time's up!");
    resetTimer();
  }
}
```

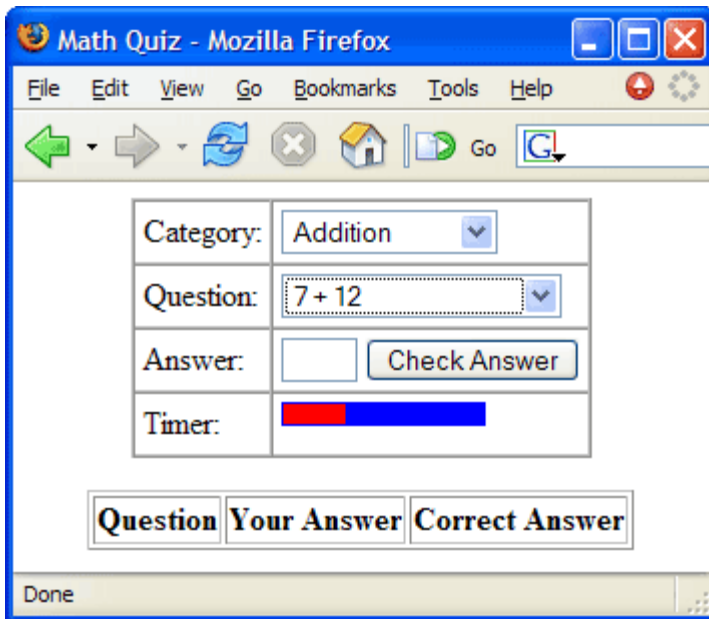
## JAVA SCRIPT

```
}  
</script>  
</head>  
  
<body onload="resetTimer();">  
  
<div id="divSliderBG"></div>  
<div id="divSlider"></div>  
  
<form>  
  <input type="button" value="Start Timer" onclick="decrementTimer();">  
</form>  
  
</body>  
</html>
```

### ***Exercise: Changing the Math Quiz Timer to a Slider***

Duration: 15 to 25 minutes.

In this exercise, you will modify the Math Quiz so that the timer is a slider rather than a count down. The result will look like this:



1. Open [DynamicHtml/Exercises/MathQuizSlider.html](#) for editing.
  2. Notice that the timer on the page has been changed from an input element to two divs.
  3. `<tr id="trTimer">`
  4. `<td>Timer:</td>`
  5. `<td>`
  6. `<div id="divSliderBG"></div>`
  8. `<div id="divSlider"></div>`
  10. `</td>`
- `</tr>`

## JAVA SCRIPT

---

11. Also notice that the `resetTimer()` function sets the width of the slider.
12. 

```
var slider = document.getElementById("divSlider");
slider.style.width = "0px";
```
13. Modify the `decrementTimer()` function as follows:
  - Create a variable `Slider` that holds the `divSlider` object.
  - Create a variable `curWidth` that holds the current width of the slider.
  - Within the `if` block add 1 to the width of the slider.
14. Test your solution in a browser.

Modify the slider so that it extends the full width of the table cell it is within.

[Where is the solution?](#)

### *Changing the Z-Index*

The z-index of an object can be changed by modifying the `zIndex` property of the element's style property. The following example demonstrates this.

#### *Code Sample: DynamicHtml/Demos/Zindex.html*

```
<html>
<head>
<title>Position</title>
<style type="text/css">
#divRed {
  position:relative;
  height:100px;
  width:100px;
  z-index:10;
  border:20px solid red;
}
#divBlue {
  position:relative;
  top:-50px;
  height:100px;
  width:100px;
  z-index:20;
  border:20px solid blue;
}
</style>
<script type="text/javascript">
var Z = 20;
function changeZ(elem) {
  Z += 10;
  var objElem = elem;
  elem.style.zIndex = Z;
}
</script>
</head>
<body>
<div id="divRed" onclick="changeZ (this) ;"></div>
<div id="divBlue" onclick="changeZ (this) ;"></div>
</body>
</html>
```

Code Explanation

---

## JAVA SCRIPT

The variable `z` always holds the highest `z`-index. The function `changeZ()` simply adds 10 to `z` and assigns the resulting value to the `zIndex` property of the passed in object.

```
var Z = 20;
function changeZ(elem){
  Z += 10;
  var objElem = elem;
  objElem.style.zIndex = Z;
}
```

### Dynamic HTML Conclusion

In this lesson of the JavaScript tutorial, you have learned to dynamically modify the content of an HTML page and to dynamically modify CSS styles of HTML elements.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

### Quick JavaScript Recap

**In this lesson of the JavaScript tutorial, you will learn...**

1. The basics of JavaScript in a context of a refresher.
2. The fundamental data types of JavaScript

3. That the DOM is not JavaScript
4. What is the main object behind AJAX

### **This is just a refresher**

The intent of this lesson is not to replace more comprehensive JavaScript classes or books. As a matter of fact, this lesson assumes you know your way around JavaScript but you may have forgotten one detail or two about the basics of the language.

Here we will go quickly over the fundamental building blocks of JavaScript and try to clarify some concepts that sometimes can be a little blurry.

### **Primitive data types**

JavaScript comes with a number of data types that we can use in our variables. Let's look at them.

#### ***Null***

Null is a data type that has only one possible value, null, which is also a reserved word in the language. We use null to represent a value that we don't know or that is missing.

```
var name = "Homer";  
var ssn = null;
```

In the above example we know what to put in the name variable but we don't know yet what to put in the ssn variable. Maybe we will know what to put in there later in our program, but maybe not.

#### ***Undefined***

The Undefined type also has a single value, undefined, and it is similar to null but not exactly the same thing.

JavaScript uses undefined as the default value for any variable that has not been initialized yet. Let's modify our previous example.

```
var name = "Homer";  
var ssn;
```

Now the value of ssn is undefined because it is no longer initialized to null or anything else.

The undefined type is used a lot when we want to detect if a global variable has been already declared. Which is kind of a code smell anyway, as we will see in an upcoming lesson.

```
//Check if we already have a start time  
if (START_TIME === undefined) {  
    START_TIME = new Date();  
}
```

# JAVA SCRIPT

---

## ***Boolean***

Boolean is a very common data type in every language. It has only two values, true and false, which are reserved words and, I hope, self-explanatory.

```
var enabled = true;
var disabled = false;
```

## ***Number***

The Number data type can represent two types of numeric values, 32-bit integers or 64-bit floating point numbers.

Number values are created using number literals, which can be in a few different formats.

```
var age = 25; // simple, decimal, integer
var price = 45.95; // floating point
var permissions = 0775; // integer in octal, 509 in decimal
                    // (note the leading zero)
var flags = 0x1c; // integer in hexadecimal, 28 in decimal
                // (note the 0x prefix)
var measurement = 5.397e-9; // floating point in
                            // scientific notation
```

## ***String***

String is a very popular data type and they are used to represent text. We spend a lot of time manipulating strings in pretty much any programming language.

We create strings using literal values enclosed in single or double quotation marks. These literal also support a few special encodings for common characters like *new line*, *tab*, and the quotation marks themselves. This is similar to what happens with strings in many other programming languages.

```
var name = 'Homer', lastName = "Simpson";
var host = 'Conan O\'Brien';
var path = 'c:\\temp\\dir\\myfile.txt';
var tabDelimited = "COL1\tCOL2\tCOL3\nVAL1\tVAL2\tVAL3";
```

Every value in JavaScript can be converted to a string by using the toString() method, like var s = myValue.toString();.

## ***Native Types***

In addition to the primitive data types we just saw, JavaScript also provides a few other data types, which are implemented as objects.

## JAVA SCRIPT

---

### ***Date***

We can store date values using Date objects. The Date object stores the date and time information internally as the number of milliseconds since January 1st 1970.

There aren't date literals in the language, so we have to explicitly create a Date object when we need one.

```
var rightNow = new Date(); // current date and time
var holiday = new Date(2008, 6, 4); // 4th of July, note the
    // 0-based month number
var birth = Date.parse('7/4/2008'); // 4th of July, format varies with browser
    // locale (avoid this)
```

There two important pitfalls in the above example: the month is a number from 0 to 11 when passed as a parameter and the parse-able string formats vary by browser implementation and by user locale, so we'd better off just avoid parsing altogether.

### ***Array***

How useful would be our programs if we couldn't organize the data in arrays or other collection structures? I guess not very much.

Arrays are very powerful in JavaScript and they kind of blur the lines between arrays and custom objects.

The Array object can be instantiated with both a constructor call or using literals. The array indices are not necessarily contiguous or numeric or even of the same data type.

```
var CITIES = new Array();
CITIES[0] = 'Albuquerque';
CITIES[9] = 'Tampa';
var TEAMS = [ 'Cubs', 'Yankees', 'Mariners' ];
var BANDS = [ ];
BANDS['rock'] = "Beatles, Rolling Stones, Pink Floyd";
BANDS['punk'] = "Sex Pistols, Ramones, Dead Kennedys";
BANDS[1992] = "Nirvana, Pearl Jam, Soundgarden, Metallica";
```

### ***Object***

The Object type serves as the base for all the objects in JavaScript, regardless of their data type or how they were created.

The Object type is also used when we want to create custom objects. We can create new objects using a constructor call or a literal. We will cover this in greater detail in a future lesson.

```
var employee = new Object();
employee.name = 'Homer Simpson';
employee.badgeNumber = 35739;
var boss = { }; //literal syntax, empty though
boss.name = 'Montgomery Burns';
```



## JAVA SCRIPT

---

```
boss.badgeNumber = 1;
employee.reportsTo = boss;
```

### ***Regular Expressions***

Regular Expressions is a syntax used to find occurrences of a string pattern inside a larger string. It has historically been more popular in Unix environments and in Perl programs, but it has gained some adoption in many other programming languages as well.

Regular expressions is one of these technologies with a measurable learning curve but that can have a big payoff depending on the type of work you do.

JavaScript implements regular expressions with RegExp objects. It also support the Perl-syle literals.

```
var text = "Webucator";
var pattern = new RegExp('cat', 'g');
var samePattern = /cat/g; //using the literal syntax
alert( pattern.test( text ) );// shows 'true'
```

### **Functions**

Functions in JavaScript are more than just static blocks of code. They are Function objects that we use just like any other data type value, e.g. we can pass functions to other functions, we can store a function in a variable, we can modify a function, etc.

We will have a lot to talk about functions in one of our lessons. For now let's just remember how we declare and call functions.

```
//declare the function
function sayHowMuch(name, price, quantity) {
  var finalPrice = price * quantity;
  alert('The price for ' + quantity + ' ' +
    name + '(s) is $' + finalPrice);
}

//call the function with arguments
sayHowMuch('ice cream cone', 1.99, 3);
sayHowMuch('Movie ticket', 10.00, 5);
```

### **The DOM is not JavaScript**

A common source of confusion is the relationship between the browser DOM (Document Object Model) and the JavaScript global objects.

JavaScript being an interpreted language with a runtime execution engine, it needs a host environment to instantiate the engine and forward the JavaScript code to it. The browser is one of many hosts for JavaScript. Other hosts are Adobe Flash plugins (via ActionScript), desktop widgets (like Yahoo! Widgets, MS Gadgets, OS X Dashboard Widgets), Firefox browser add-ons, and even some kinds of electronic equipment.

## JAVA SCRIPT

---

All that said, the browser was the originally intended host for JavaScript and by far the most common one.

### ***The DOM***

It's important to understand that the DOM is a standard that has nothing to do with JavaScript. It was created by the W3C to normalize the browser vendors' implementations of Dynamic HTML.

The DOM is an API that enables programatic access to the HTML document structure, for reading or modification purposes. When we write code like `document.getElementById('abc')` we are using the DOM.

With the DOM we can traverse our entire HTML document looking for specific HTML elements, which are called *nodes*, or even create new elements and append them pretty much anywhere we want inside the HTML document.

### ***The window object***

In browser scripts, the document object is actually a property of the window object, which is the default (or global) object of JavaScript in that environment. So typing `window.document.body` is the same as typing `document.body`. The DOM starts at the document object.

There are other things one may think are part of JavaScript when, in fact, they're browser-specific features, like the `alert()`, `prompt()`, `setTimeout()`, and `open()` functions. These are just methods of the window object, not part of JavaScript per se.

### **The XMLHttpRequest object**

Another important object that we use a lot in JavaScript these days is the XMLHttpRequest object. This is the object that powers the AJAX functionality in a lot of web pages.

This object is also *not* part of JavaScript. It can be used from JavaScript but it isn't part of the language.

We won't cover AJAX here but it suffices to understand that this object allows our scripts to initiate a request to an URL and collect the server response without the need to reload the entire page.

```
function saveUser(name, age) {
  //the following line is actually a grossly simplified version
  // that is not supported in all browsers
  var ajax = new XMLHttpRequest();

  ajax.open('POST', '/users/update', true);
  ajax.onreadystatechange = function () {
    if (ajax.readyState == 4) {
      alert('User ' + name + ' updated.');
```

## JAVA SCRIPT

---

```
ajax.send( 'userName=' + name + '?userAge=' + age );  
}  
saveUser('Joe Doe', 54);
```

### Quick JavaScript Recap Conclusion

We hope that after this short tour of JavaScript data types and the browser execution environment you will be able to absorb all the new concepts you are about to see in the remainder of the course.

Because of the syntax similarities with C-style languages, it's often possible that we mix-up what is available in JavaScript so this lesson can serve as a quick reminder or cheat-sheet when needed.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.

### Advanced Techniques

**In this lesson of the JavaScript tutorial, you will learn...**

1. To use the default operator.
2. To pass a flexible number of arguments to a function.

3. To pass a function as an argument to another function.
4. To create anonymous functions.
5. Other techniques related to functions.

## Beyond The Basics

You can go very far in JavaScript using only the language basics that we have seen up to now but, to really unleash JavaScript's power, we need to get more familiar with a few programming techniques.

These techniques will help your code become more expressive and more reusable at the same time. They are programming patterns that leverage some language features that may not be present in other programming languages.

If you don't have experience with any other dynamically typed or functional language, the patterns you will see here may look very strange at first. But fear not, we will provide plenty of explanations and examples to make you feel comfortable enough to read and write JavaScript code that employ what you just learned.

## Optional Function Arguments

When we declare a function in JavaScript, we normally include a list of the arguments the function expects.

### *Code Sample: AdvancedTechniques/Demos/sumAll-1.html*

```
---- Code Omitted ----
```

```
function sumValues(val1, val2, val3) {  
  return val1 + val2 + val3;  
}
```

```
---- Code Omitted ----
```

But this does not guarantee that our function will always be called with 3 arguments. It's perfectly valid for someone to call our function passing more than 3 or even less than 3 arguments.

```
var R1 = sumValues(3, 5, 6, 2, 7);  
var R2 = sumValues(12, 20);
```

Both calls will return surprising results (surprising from the caller's perspective.)

In the first case, since we are not expecting more than 3 arguments, the extra values, 2 and 7, will simply be ignored. It's bad because the returned value is probably not what the calling code expected.

It's even worse when we look at the second example. We are passing only 2 arguments. What happens to val3? It will have the value of undefined. This will cause the resulting sum to be NaN, which is clearly undesirable.

## JAVA SCRIPT

Let's fix our function to deal with these types of situations.

### ***Code Sample: AdvancedTechniques/Demos/sumAll-2.html***

```
---- Code Omitted ----

function sumValues(val1, val2, val3) {
  if (val1 === undefined) {
    val1 = 0;
  }

  if (val2 === undefined) {
    val2 = 0;
  }

  if (val3 === undefined) {
    val3 = 0;
  }

  return val1 + val2 + val3;
}

var R1 = sumValues(3, 5, 6, 2, 7);
var R2 = sumValues(12, 20);
---- Code Omitted ----
```

If we run our example again, we will see that we no longer get NaN for the second function call. Instead we get 32, which is probably what the calling code expected.

We now have a pretty robust function that adds 3 numbers but it still doesn't feel all that useful. Sooner or later we will need to add four or five numbers and we don't want to be updating our function to accept a val4 then a val5 parameters. That would be a less than desirable maintenance task. Fortunately JavaScript can help us with that too.

Every function, when called, receives a hidden parameter called arguments, which is an array of all the arguments passed to the function. Back to our example, the first time we call sumValues, the arguments array will contain [3, 5, 6, 2, 7] and in the second call [12, 20].

What this means is that we can ignore the passed parameters altogether and deal only with the arguments array. Let's update our function once again.

### ***Code Sample: AdvancedTechniques/Demos/sumAll-3.html***

```
---- Code Omitted ----

function sumValues() {
  var sum = 0;
  for (var i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

## JAVA SCRIPT

```
var R1 = sumValues(3, 5, 6, 2, 7);
var R2 = sumValues(12, 20);
---- Code Omitted ----
```

Note how we got rid of the parameter list and now we get the values directly from the arguments array. When we run our example now we see that the returned values are correct and precisely what was expected by the caller. We now have a function that accepts as many parameters as are thrown at it and will always return the sum of all those arguments.

## Truthy and Falsy

JavaScript, as we already know, has a boolean data type, which has only two possible values: true or false. Boolean expressions also evaluate to a boolean value. But that's not the entire story.

When used in a context that expects a boolean value, any JavaScript expression can be used. See below.

```
var NUMBER = 0;
if (NUMBER) {
  alert('You should not see this');
}
NUMBER = 1;
if (NUMBER) {
  alert('You should be reading this');
}
var TEXT;
if (TEXT) {
  alert('You should not see this');
}
TEXT = "";
if (TEXT) {
  alert('You should not see this');
}
TEXT = "hi";
if (TEXT) {
  alert('You should be reading this');
}
```

In the example above we are using non-boolean expressions (NUMBER and TEXT) in the if statement and some of these expressions are being understood as true and some others false.

What is happening here is *Type Coercion*. JavaScript does its best to convert the given expression into the desired type (boolean.)

JavaScript resolves the following values to false.

- false (of course)
- null
- undefined
- 0 (zero)
- NaN
- "" (empty string)

## JAVA SCRIPT

---

The above values are all referred to as *falsy*, which is a way of saying "Not the same as false but can be interpreted like such."

Every other value, including the strings "0" and "false", will resolve to true and will be referred to as *truthy*. Again, "Not the same as true but can be interpreted as such."

Type coercion is the reason we have the === (triple-equal or strictly equal) comparison operator in JavaScript. The regular equality operator == applies type coercion and sometimes your comparisons will not result as expected. Look at the following sample code.

```
var NUM = 0;
if (NUM == "") {
  alert('Hey, I did not expect to see this.');
```

```
}
if (NUM === "") {
  alert('This will not be displayed.');
```

```
}
```

In the first if conditional is comparing two *falsy* values, and the type coercion will resolve both of them to false, causing the result of the comparison to be true, which is probably not the original intent of the code.

### Default Operator

To detect the type difference (string vs. number) we would need to use the triple equal operator, as shown in the second if statement.

The boolean operators && and || also use truthy and falsy to resolve each of the operands to a boolean value.

From your previous experiences with other programming languages you may be led to believe that the result of a boolean operation is always true and false. This is not the case in JavaScript.

In JavaScript, the result of a boolean operation is the value of the operand that determined the result of the operation. Let's clarify that with an example.

```
var A = 0, B = NaN, C = 1, D = "hi";
var RESULT = ( A || B || C || D );
alert("Result = " + RESULT);
```

```
RESULT = ( D && C && B && A );
alert("Result = " + RESULT);
```

The first boolean expression ( A || B || C || D ) is evaluated from left to right until a conclusion is made. The || is the boolean OR operator, which only needs one of the operands to be true for the operation result in true. A is 0, which is falsy. The evaluation continues with the remaining operands because falsy didn't determine anything yet. When checking B, which is NaN and also falsy, we have the same situation - we need to continue evaluating. Then we check C, which is 1 and resolves to true. We no longer need to continue evaluating the remaining operands because we already know the expression will result true. But here's the catch. Instead of resulting strictly

## JAVA SCRIPT

---

true, it will result in the truthy value that ended the evaluation, C in our case. The message displayed will be "Result = 1".

As you might already expect, the `&&` (boolean AND operator) works in a similar fashion, but with opposite conditions. The AND operator returns false as soon as it finds an operand that is false. If for the expression ( `D && C && B && A` ) we follow the same sequence we did for the OR operator we will see that, D and C are both truthy so we need to keep going, then we get to B, which is falsy and causes the evaluation to stop and return B. The message displayed then is "Result = NaN".

You may be reading all this and thinking how can this be of any use. It turns out that this behavior of returning the first conclusive value can be very handy when ensuring that a variable is initialized.

Let's take another look at a function we saw back in the Form Validation lesson.

```
function checkLength(text, min, max){
  min = min || 1;
  max = max || 10000;

  if (text.length < min || text.length > max) {
    return false;
  }
  return true;
}
```

The first two lines in this function make sure that min and max always have a valid value. This allow the function to be called like `checkValue("abc")`. In this case the min and max parameters will both start with the undefined value.

When we reach the line `min = min || 1;` we are simply assigning 1 to min, ensuring it overrides the undefined. Similarly we assign 1000 to max.

If we had passed actual values for these parameters as in `checkLength("abc", 2, 10)` these values would be kept because they are truthy.

With this usage of the `||` we are effectively providing default values for these two parameters. That's why this operator, in this context, is also called the *Default Operator*.

The default operator replaces more verbose code like:

```
if (min === undefined) {
  min = 1;
}
// becomes simply
min = min || 1;

var contactInfo;
if (email) {
  contactInfo = email;
} else if (phone) {
  contactInfo = phone;
}
```



## JAVA SCRIPT

---

```
} else if (streetAddress) {  
    contactInfo = streetAddress;  
}  
// is greatly shortened to  
var contactInfo = email || phone || streetAddress;
```

### ***Exercise: Applying defaults to function parameters***

Duration: 15 to 25 minutes.

Here we will revisit an earlier example and use the default operator to handle optional parameters.

1. Open `AdvancedTechniques/Exercises/sumAll-defaults.html` for editing.
2. Edit the existing `sumAll()` to use the default operator instead of the if blocks.

### ***Code Sample: AdvancedTechniques/Exercises/sumAll-defaults.html***

```
<html>  
<head>  
  <title>Sum all numbers, default operator</title>  
  <style type="text/css">  
    .wc_debug  
    {  
      background-color:#ffc;  
    }  
  </style>  
</head>  
<body>  
  <h1>Sum all numbers, default operator</h1>  
  <script type="text/javascript" src="../../Libraries/DebugHelp.js" ></script>  
  <script type="text/javascript">  
    insertDebugPanel();  
  
    // this function uses the if blocks to  
    // handle the optional parameters.  
    // Change it to use the default operator instead.  
  
    function sumValues(val1, val2, val3) {  
      if (val1 === undefined) {  
        val1 = 0;  
      }  
  
      if (val2 === undefined) {  
        val2 = 0;  
      }  
  
      if (val3 === undefined) {  
        val3 = 0;  
      }  
  
      return val1 + val2 + val3;  
    }  
  
    var R1 = sumValues(3, 5, 6, 2, 7);  
    var R2 = sumValues(12, 20);
```

## JAVA SCRIPT

```
//print the results
debugWrite(R1);
debugWrite(R2);

</script>

</body>
</html>
```

[Where is the solution?](#)

## Functions Passed as Arguments

In JavaScript functions are first class data types. Functions aren't just an immutable block of code that can only be invoked. In JavaScript each function we declare becomes an object, with its own properties and methods, and can also be passed around like any other object.

Let's see how we can use functions as parameters to other functions. Consider the following example.

### ***Code Sample: AdvancedTechniques/Demos/function-arguments.html***

```
---- Code Omitted ----

var VALUES = [5, 2, 11, -7, 1];

function sum(a, b) {
  return a+b;
}

function multiply(a, b) {
  return a*b;
}

function combineAll(list, initialValue, operation) {
  var runningResult = initialValue;
  for (var i=0; i< list.length; i++) {
    runningResult = operation(runningResult, list[i]);
  }
  return runningResult;
}

var SUM = combineAll(VALUES, 0, sum);
var PRODUCT = combineAll(VALUES, 1, multiply);
---- Code Omitted ----
```

You may be wondering what the following function call means: `var SUM = combineAll(VALUES, sum);`. In this statement we are passing the function `sum` as the second parameter of `combineAll`. We are not *invoking* `sum` yet, just passing a reference to it. Note that the open and close parenthesis aren't used after `sum`, that should serve as a tip off that this is not a function invocation.

## JAVA SCRIPT

The line that ultimately invokes `sum` is `runningResult = operation(initialValue, list[i]);`, which received a reference to `sum` in the `operation` parameter. When `operation` is invoked, in reality, it is `sum` that is getting called, returning the sum of the two values passed in.

This is a very important technique and the `combineAll` function is often called *reduce*. Take your time to review the code and run the example until you feel comfortable with it. We will be using this capability extensively in the remaining lessons.

### Anonymous Functions

Going back to our previous example, the functions `sum` and `multiply` are only referred to once, in each call to `combineAll`. Furthermore, if we stick to that pattern, any new combination behavior that we desire, such as concatenate the values or compute the average value, will need a new function just to be passed to `combineAll`. That seems like too much overhead for such a simple thing. It would also not be very interesting to have all these functions that do such simple things scattered through out the code.

Thankfully, we don't actually need to declare each of these functions. We don't even need to come up with names for them. JavaScript allow us to create functions *on the spot*, any time we need a function that will only be used at that spot.

The syntax is rather compact.

#### Syntax

```
function (arg1, arg2) {  
  //function statements here  
}
```

Because the functions created this way don't have names, they are aptly called *anonymous functions*.

Let's revisit our previous example and use anonymous functions to replace the single-use functions we declared.

#### ***Code Sample: AdvancedTechniques/Demos/anon-func-arguments.html***

```
---- Code Omitted ----  
  
var VALUES = [5, 2, 11, -7, 1];  
  
function combineAll(list, initialValue, operation) {  
  var runningResult = initialValue;  
  for (var i=0; i< list.length; i++) {  
    runningResult = operation(runningResult, list[i]);  
  }  
  return runningResult;  
}  
  
var SUM = combineAll(VALUES, 0, function (a, b) {  
  return a+b;  
});
```

## JAVA SCRIPT

```
var PRODUCT = combineAll(VALUEES, 1, function (a, b) {  
    return a*b;  
});  
---- Code Omitted ----
```

The highlighted code represent the two anonymous functions, located where previously we had sum and multiply. This coding style can understandably be harder to read, but it also avoids all that jumping around to look up what that function that you are passing by name really does. The code of that function is right there, next to the code that is using it.

## Inner Functions

Since functions in JavaScript are just one more type of object, we can create a function inside another function. These are called *inner functions*.

The example below shows how to create and use a function inside another one.

```
function analyzeText(text) {  
    var index = 0;  
  
    function getNextCharacter() {  
        if (index < index.length) {  
            return text.charAt(index);  
        }  
        return false;  
    }  
  
    var c = getNextCharacter();  
    while (c) {  
        alert(index + ' ---> ' + c );  
        c = getNextCharacter();  
    }  
}  
analyzeText('abcdef');
```

The above example is not particularly useful. We will see more important uses of inner function when we look at private members. For the time being, just notice how `getNextCharacter()` has access to `index` and `text`, which are scoped to the `analyzeText()` function.

## The eval() Function

The reason we are mentioning `eval()` in this lesson is to acknowledge its existence and to urge you not to use it. We will explain why, but first let's explain what it does.

`eval` interprets a string containing JavaScript code. It can be a simple expression like `"1 + 2"` or a long and complex script, with functions and all.

Here's one example that is not too different from what we can find in live sites on the web.

```
function getProperty(objectName, propertyName) {  
    var expression = objectName + "." + propertyName;
```

## JAVA SCRIPT

---

```
var propertyValue = eval(expression);
return propertyValue;
}
var PROP = "title"; //assume this was given by the user
alert(getProperty("document", PROP)); //shows the window title
```

This function creates a JavaScript expression by concatenating an object name, with a dot and a property name. Then it uses eval to evaluate that expression.

As we can see eval is a powerful function, but it is also potentially dangerous and incredibly inefficient. It's dangerous because it's typically used to evaluate user entered input, which not always is a safe thing to do. It's inefficient because *each call to eval starts a JavaScript compiler*.

The use of eval normally reveals lack of knowledge from the developer that wrote the script. For example, the sample that we just used is not necessary. Probably what happened was that the developer did not know about the [ ] accessor for properties. The same effect would be obtained with alert(window[PROP]);, with the advantage of not firing up a compiler just to retrieve the property value.

Remember this, *eval is evil*. Avoid it as much as you can. If you think you need it, maybe it's because you did not learn yet about an alternative way in JavaScript.

## Variable Scope

Variable scope defines which parts of your code have access to a variable that you define. This typically varies depending where you declare the variable.

Variables in JavaScript are either *global* or *function scoped*. When a variable is declared outside of any function body, then it will become a global variable, meaning that *any* portion of your script will have access to it.

```
var NAME = 'my global value';
function displayName() {
    alert(NAME);
}
alert(NAME);
displayName();
```

The previous sample showed that the Name variable is visible inside the function displayName.

There's a catch, though. If you forget to declare the variable using the var operator before using the variable, the variable *will be created as a global variable* even if you are declaring it inside a function or inside a for loop declaration.

```
function prepare() {
    TEST = 123;
    alert(typeof TEST);
    //let's forget the "var" in the "for" declaration
    for (abc=0; abc<5; abc++) {
        //...
    }
}
```

---

```
alert (typeof TEST);  
prepare ();  
alert (TEST);  
alert (abc);
```

## ***Function Scope***

Variables declared with var inside a function will be visible only inside that function. It doesn't matter if the variable was declared inside an if block or a for block. Once declared, the variable becomes visible throughout the remainder of the function. What that means is that JavaScript doesn't have block scope.

## **Advanced Techniques Conclusion**

Hopefully, with what you learned in this lesson you will be able to write much more robust JavaScript code and start leveraging some of the flexibility JavaScript puts at your disposal to create really powerful code.

If you still feel unsure of how these techniques work, re-read the lesson and look for alternative explanations on the web. Sometimes it helps reading a different phrasing of the same topic. It's important that you can at least read this type of code fluently in order to understand some of what we will be looking at in the upcoming lessons.

To continue to learn JavaScript go to the [top of this page](#) and click on the next lesson in this JavaScript Tutorial's Table of Contents.