Javascript Tutorial

Tutorialspoint.com

***Javascript* is a scripting language produced by Netscape for use within HTML Web pages.**

JavaScript is loosely based on Java and it is built into all the major modern browsers. This tutorial gives an initial push to start you with Javascript. For more detail kindly check tutorialspoint.com/javascript

What is JavaScript ?

JavaScript is:

- JavaScript is a lightweight, interpreted programming language
- Designed for creating network-centric applications
- Complementary to and integrated with Java
- Complementary to and integrated with HTML
- Open and cross-platform

JavaScript Syntax:

A JavaScript consists of JavaScript statements that are placed within the <script>... </script> HTML tags in a web page.

You can place the <script> tag containing your JavaScript anywhere within you web page but it is preferred way to keep it within the <head> tags.

The <script> tag alert the browser program to begin interpreting all the text between these tags as a script. So simple syntax of your JavaScript will be as follows

```
<script ...>
  JavaScript code
</script>
```

The script tag takes two important attributes:

- **language:** This attribute specifies what scripting language you are using. Typically, its value will be *javascript*. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
- **type:** This attribute is what is now recommended to indicate the scripting language in use and its value should be set to *"text/javascript"*.

So your JavaScript segment will look like:

```
<script language="javascript" type="text/javascript">
  JavaScript code
</script>
```

Your First JavaScript Script:

Let us write our class example to print out "Hello World".

```
<html>
<body>
<script language="javascript" type="text/javascript">
<!--
   document.write("Hello World!")
//-->
</script>
</body>
</html>
```

Above code will display following result:

```
Hello World!
```

Whitespace and Line Breaks:

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs.

Because you can use spaces, tabs, and newlines freely in your program so you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

Semicolons are Optional:

Simple statements in JavaScript are generally followed by a semicolon character, just as they are in C, C++, and Java. JavaScript, however, allows you to omit this semicolon if your statements are each placed on a separate line. For example, the following code could be written without semicolons

```
<script language="javascript" type="text/javascript">
<!--
  var1 = 10
  var2 = 20
//-->
</script>
```

But when formatted in a single line as follows, the semicolons are required:

```
<script language="javascript" type="text/javascript">
<!--
  var1 = 10; var2 = 20;
//-->
</script>
```

Note: It is a good programming practice to use semicolons.

Case Sensitivity:

JavaScript is a case-sensitive language. This means that language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters.

So identifiers *Time*, *TIme* and *TIME* will have different meanings in JavaScript.

NOTE: Care should be taken while writing your variable and function names in JavaScript.

Comments in JavaScript:

JavaScript supports both C-style and C++-style comments, Thus:

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is treated as a comment. This may span multiple lines.
- JavaScript also recognizes the HTML comment opening sequence <!--. JavaScript treats this as a single-line comment, just as it does the // comment.
- The HTML comment closing sequence --> is not recognized by JavaScript so it should be written as //-->.

JavaScript Placement in HTML File:

There is a flexibility given to include JavaScript code anywhere in an HTML document. But there are following most preferred ways to include JavaScript in your HTML file.

- Script in <head>...</head> section.
- Script in <body>...</body> section.
- Script in <body>...</body> and <head>...</head> sections.
- Script in and external file and then include in <head>...</head> section.

JavaScript DataTypes:

JavaScript allows you to work with three primitive data types:

- Numbers eg. 123, 120.50 etc.
- Strings of text e.g. "This text string" etc.
- Boolean e.g. true or false.

JavaScript also defines two trivial data types, *null* and *undefined*, each of which defines only a single value.

JavaScript Variables:

Like many other programming languages, JavaScript has variables. Variables can be thought of as named containers. You can place data into these containers and then refer to the data simply by naming the container.

Before you use a variable in a JavaScript program, you must declare it. Variables are declared with the **var** keyword as follows:

```
<script type="text/javascript">
<!--
var money;
var name;
//-->
</script>
```

JavaScript Variable Scope:

The scope of a variable is the region of your program in which it is defined. JavaScript variable will have only two scopes.

- **Global Variables:** A global variable has global scope which means it is defined everywhere in your JavaScript code.
- **Local Variables:** A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

JavaScript Variable Names:

While naming your variables in JavaScript keep following rules in mind.

- You should not use any of the JavaScript reserved keyword as variable name. These keywords are mentioned in the next section. For example, *break* or *boolean* variable names are not valid.
- JavaScript variable names should not start with a numeral (0-9). They must begin with a letter or the underscore character. For example, *123test* is an invalid variable name but *_123test* is a valid one.
- JavaScript variable names are case sensitive. For example, *Name* and *name* are two different variables.

JavaScript Reserved Words:

The following are reserved words in JavaScript. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

abstract	else	instanceof	switch
boolean	enum	int	synchronized
break	export	interface	this
byte	extends	long	throw
case	false	native	throws
catch	final	new	transient
char	finally	null	true
class	float	package	try
const	for	private	typeof
continue	function	protected	var
debugger	goto	public	void
default	if	return	volatile
delete	implements	short	while
do	import	static	with
double	in	super	

The Arithmatic Operators:

There are following arithmatic operators supported by JavaScript language:

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200
/	Divide numerator by denumerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0

++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

The Comparison Operators:

There are following comparison operators supported by JavaScript language

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Logical Operators:

There are following logical operators supported by JavaScript language

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

The Bitwise Operators:

There are following bitwise operators supported by JavaScript language

Assume variable A holds 2 and variable B holds 3 then:

Operator	Description	Example
&	Called Bitwise AND operator. It performs a Boolean AND operation on each bit of its integer arguments.	(A & B) is 2 .
|	Called Bitwise OR Operator. It performs a Boolean OR operation on each bit of its integer arguments.	(A | B) is 3.
^	Called Bitwise XOR Operator. It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both.	(A ^ B) is 1.
~	Called Bitwise NOT Operator. It is a is a unary operator and operates by reversing all bits in the operand.	(~B) is -4 .
<<	Called Bitwise Shift Left Operator. It moves all bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying by 2, shifting two positions is equivalent to multiplying by 4, etc.	(A << 1) is 4.
>>	Called Bitwise Shift Right with Sign Operator. It moves all bits in its first operand to the right by the number of places specified in the second operand. The bits filled in on the left depend on the sign bit of the original operand, in order to preserve the sign of the result. If the first operand is positive, the result has zeros placed in the high bits; if the first operand is negative, the result has ones placed in the high bits. Shifting a value right one place is equivalent to dividing by 2 (discarding the remainder), shifting right two places is equivalent to integer division by 4, and so on.	(A >> 1) is 1.
>>>	Called Bitwise Shift Right with Zero Operator. This operator is just like the >> operator, except that the bits shifted in on the left are always zero,	(A >>> 1) is 1.

The Assignment Operators:

There are following assignment operators supported by JavaScript language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies	C *= A is equivalent to C = C *

	right operand with the left operand and assign the result to left operand	A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A

Miscellaneous Operator

The Conditional Operator (? :)

There is an oprator called conditional operator. This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation. The conditioanl operator has this syntax:

Operator	Description	Example
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

The *typeof* Operator

The *typeof* is a unary operator that is placed before its single operand, which can be of any type. Its value is a string indicating the data type of the operand.

The *typeof* operator evaluates to "number", "string", or "boolean" if its operand is a number, string, or boolean value and returns true or false based on the evaluation.

if statement:

The **if** statement is the fundamental control statement that allows JavaScript to make decisions and execute statements conditionally.

Syntax:

```
if (expression){
   Statement(s) to be executed if expression is true
}
```

if...else statement:

The **if...else** statement is the next form of control statement that allows JavaScript to execute statements in more controlled way.

Syntax:

```
if (expression){
   Statement(s) to be executed if expression is true
}else{
   Statement(s) to be executed if expression is false
}
```

if...else if... statement:

The **if...else if...** statement is the one level advance form of control statement that allows JavaScript to make correct decision out of several conditions.

Syntax:

```
if (expression 1){
   Statement(s) to be executed if expression 1 is true
}else if (expression 2){
   Statement(s) to be executed if expression 2 is true
}else if (expression 3){
   Statement(s) to be executed if expression 3 is true
}else{
   Statement(s) to be executed if no expression is true
}
```

switch statement:

The basic syntax of the **switch** statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each **case** against the value of the expression until a match is found. If nothing matches, a **default** condition will be used.

```
switch (expression)
{
  case condition 1: statement(s)
                    break;
  case condition 2: statement(s)
                    break;
   ...
  case condition n: statement(s)
                    break;
  default: statement(s)
}
```

The *while* Loop

The most basic loop in JavaScript is the **while** loop which would be discussed in this tutorial.

Syntax:

```
while (expression){
   Statement(s) to be executed if expression is true
}
```

The *do...while* Loop:

The **do...while** loop is similar to the **while** loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once, even if the condition is *false*.

Syntax:

```
do{
   Statement(s) to be executed;
} while (expression);
```

The *for* Loop

The **for** loop is the most compact form of looping and includes the following three important parts:

- The loop initialization where we initialize our counter to a starting value. The initialization statement is executed before the loop begins.
- The test statement which will test if the given condition is true or not. If condition is true then code given inside the loop will be executed otherwise loop will come out.
- The iteration statement where you can increase or decrease your counter.

You can put all the three parts in a single line separated by a semicolon.

Syntax:

```
for (initialization; test condition; iteration statement){
     Statement(s) to be executed if test condition is true
}
```

The *for...in* Loop

```
for (variablename in object){
  statement or block to execute
}
```

In each iteration one property from *object* is assigned to *variablename* and this loop continues till all the properties of the object are exhausted.

The *break* Statement:

The **break** statement, which was briefly introduced with the *switch* statement, is used to exit a loop early, breaking out of the enclosing curly braces.

The *continue* Statement:

The **continue** statement tells the interpreter to immediately start the next iteration of the loop and skip remaining code block.

When a **continue** statement is encountered, program flow will move to the loop check expression immediately and if condition remain true then it start next iteration otherwise control comes out of the loop.

Function Definition:

Before we use a function we need to define that function. The most common way to define a function in JavaScript is by using the function keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces. The basic syntax is shown here:

```
<script type="text/javascript">
```

```
<!--
function functionname(parameter-list)
{
  statements
}
//-->
</script>
```

Calling a Function:

To invoke a function somewhere later in the script, you would simple need to write the name of that function as follows:

```
<script type="text/javascript">

<!--
sayHello();
//-->
</script>
```

Exceptions

Exceptions can be handled with the common try/catch/finally block structure.

```
<script type="text/javascript">
<!--
try {
   statementsToTry
} catch ( e ) {
      catchStatements
} finally {
      finallyStatements
}
//-->
</script>
```

The try block must be followed by either exactly one catch block or one finally block (or one of both). When an exception occurs in the catch block, the exception is placed in e and the catch block is executed. The finally block executes unconditionally after try/catch.

Alert Dialog Box:

An alert dialog box is mostly used to give a warning message to the users. Like if one input field requires to enter some text but user does not enter that field then as a part of validation you can use alert box to give warning message as follows:

```
<head>
<script type="text/javascript">
<!--
   alert("Warning Message");
//-->

</script>
</head>
```

Confirmation Dialog Box:

A confirmation dialog box is mostly used to take user's consent on any option. It displays a dialog box with two buttons: **OK** and **Cancel**.

You can use confirmation dialog box as follows:

```
<head>
<script type="text/javascript">
<!--
   var retVal = confirm("Do you want to continue ?");
   if( retVal == true ){
      alert("User wants to continue!");
          return true;
   }else{
      alert("User does not want to continue!");
          return false;
   }
//-->
</script>
</head>
```

Prompt Dialog Box:

You can use prompt dialog box as follows:

```
<head>
<script type="text/javascript">
<!--
   var retVal = prompt("Enter your name : ", "your name here");
   alert("You have entered : " +  retVal );
//-->
</script>
</head>
```

Page Re-direction

This is very simple to do a page redirect using JavaScript at client side. To redirect your site visitors to a new page, you just need to add a line in your head section as follows:

```
<head>
<script type="text/javascript">
<!--
   window.location="http://www.newlocation.com";
//-->
</script>
</head>
```

The void Keyword:

The **void** is an important keyword in JavaScript which can be used as a unary operator that appears before its single operand, which may be of any type.

This operator specifies an expression to be evaluated without returning a value. Its syntax could be one of the following:

```
<head>
<script type="text/javascript">
<!--
void func()
javascript:void func()

or:

void(func())
javascript:void(func())
//-->
</script>
</head>
```

The Page Printing:

JavaScript helps you to implement this functionality using **print** function of *window* object.

The JavaScript print function **window.print()** will print the current web page when executed. You can call this function directly using *onclick* event as follows:

```
<head>
<script type="text/javascript">
<!--
//-->
</script>
</head>
<body>
<form>
<input type="button" value="Print" onclick="window.print()" />
</form>
</body>
```

Storing Cookies:

The simplest way to create a cookie is to assign a string value to the *document.cookie* object, which looks like this:

Syntax:

```
document.cookie = "key1=value1;key2=value2;expires=date";
```

Reading Cookies:

Reading a cookie is just as simple as writing one, because the value of the *document.cookie* object is the cookie. So you can use this string whenever you want to access the cookie.

The *document.cookie* string will keep a list of *name=value* pairs separated by semicolons, where *name* is the *name* of a cookie and value is its string value.

JavaScript - Page Redirection

What is page redirection ?

When you click a URL to reach to a page X but internally you are directed to another page Y that simply happens because of page re-direction. This concept is different from JavaScript Page Refresh.

There could be various reasons why you would like to redirect from original page. I'm listing down few of the reasons:

- You did not like the name of your domain and you are moving to a new one. Same time you want to direct your all visitors to new site. In such case you can maintain your old domain but put a single page with a page re-direction so that your all old domain visitors can come to your new domain.
- You have build-up various pages based on browser versions or their names or may be based on different countries, then instead of using your server side page redirection you can use client side page redirection to land your users on appropriate page.
- The Search Engines may have already indexed your pages. But while moving to another domain then you would not like to lose your visitors coming through search engines. So you can use client side page redirection. But keep in mind this should not be done to make search engine a fool otherwise this could get your web site banned.

How Page Re-direction works ?

Example 1:

This is very simple to do a page redirect using JavaScript at client side. To redirect your site visitors to a new page, you just need to add a line in your head section as follows:

```
<head>
<script type="text/javascript">
<!--
   window.location="http://www.newlocation.com";
//-->
</script>
</head>
```

To understand it in better way you can Try it yourself.

Example 2:

You can show an appropriate message to your site visitors before redirecting them to a new page. This would need a bit time delay to load a new page. Following is the simple example to implement the same:

```
<head>
<script type="text/javascript">
<!--
function Redirect()
{
    window.location="http://www.newlocation.com";
}

document.write("You will be redirected to main page in 10 sec.");
setTimeout('Redirect()', 10000);
//-->
</script>
</head>
```

Here *setTimeout()* is a built-in JavaScript function which can be used to execute another function after a given time interval.

To understand it in better way you can Try it yourself.

Example 3:

Following is the example to redirect site visitors on different pages based on their browsers :

```
<head>
<script type="text/javascript">
<!--
var browsername=navigator.appName;
if( browsername == "Netscape" )
{
   window.location="http://www.location.com/ns.htm";
}
else if ( browsername =="Microsoft Internet Explorer")
{
   window.location="http://www.location.com/ie.htm";
}
else
{
  window.location="http://www.location.com/other.htm";
}
//-->
</script>
</head>
```

JavaScript - Errors & Exceptions Handling

There are three types of errors in programming: (a) Syntax Errors and (b) Runtime Errors (c) Logical Errors:

Syntax errors:

Syntax errors, also called parsing errors, occur at compile time for traditional programming languages and at interpret time for JavaScript.

For example, the following line causes a syntax error because it is missing a closing parenthesis:

```
<script type="text/javascript">
<!--
window.print(;
//-->
</script>
```

When a syntax error occurs in JavaScript, only the code contained within the same thread as the syntax error is affected and code in other threads gets executed assuming nothing in them depends on the code containing the error.

Runtime errors:

Runtime errors, also called exceptions, occur during execution (after compilation/interpretation).

For example, the following line causes a run time error because here syntax is correct but at run time it is trying to call a non existed method:

```
<script type="text/javascript">
<!--
window.printme();
//-->
</script>
```

Exceptions also affect the thread in which they occur, allowing other JavaScript threads to continue normal execution.

Logical errors:

Logic errors can be the most difficult type of errors to track down. These errors are not the result of a syntax or runtime error. Instead, they occur when you make a mistake in the logic that drives your script and you do not get the result you expected.

You can not catch those errors, because it depends on your business requirement what type of logic you want to put in your program.

The *try...catch...finally* Statement:

The latest versions of JavaScript added exception handling capabilities. JavaScript implements the **try...catch...finally** construct as well as the **throw** operator to handle exceptions.

You can *catch* programmer-generated and *runtime* exceptions, but you cannot *catch* JavaScript syntax errors.

Here is the **try...catch...finally** block syntax:

```
<script type="text/javascript">
<!--
try {
    // Code to run
    [break;]
} catch ( e ) {
    // Code to run if an exception occurs
    [break;]
}[ finally {
    // Code that is always executed regardless of
    // an exception occurring
}]
//-->
</script>
```

The **try** block must be followed by either exactly one **catch** block or one **finally** block (or one of both). When an exception occurs in the **try** block, the exception is placed in **e** and the **catch** block is executed. The optional **finally** block executes unconditionally after try/catch.

Examples:

Here is one example where we are trying to call a non existing function this is causing an exception raise. Let us see how it behaves without with **try...catch**:

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   alert("Value of variable a is : " + a );

}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

To understand it in better way you can Try it yourself.

Now let us try to catch this exception using **try...catch** and display a user friendly message. You can also suppress this message, if you want to hide this error from a user.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   try {
      alert("Value of variable a is : " + a );
   } catch ( e ) {
      alert("Error: " + e.description );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

To understand it in better way you can Try it yourself.

You can use **finally** block which will always execute unconditionally after try/catch. Here is an example:

```
<html>
```

```
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;

   try {
      alert("Value of variable a is : " + a );
   }catch ( e ) {
      alert("Error: " + e.description );
   }finally {
      alert("Finally block will always execute!" );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

To understand it in better way you can Try it yourself.

The *throw* Statement:

You can use **throw** statement to raise your built-in exceptions or your customized exceptions. Later these exceptions can be captured and you can take an appropriate action.

Following is the example showing usage of **throw** statement.

```
<html>
<head>
<script type="text/javascript">
<!--
function myFunc()
{
   var a = 100;
   var b = 0;

   try{
      if ( b == 0 ){
         throw( "Divide by zero error." );
      }else{
         var c = a / b;
      }
   }catch ( e ) {
      alert("Error: " + e );
   }
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
```

```
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

To understand it in better way you can Try it yourself.

You can raise an exception in one function using a string, integer, Boolean or an object and then you can capture that exception either in the same function as we did above, or in other function using **try...catch** block.

The *onerror()* Method

The **onerror** event handler was the first feature to facilitate error handling for JavaScript. The **error**event is fired on the window object whenever an exception occurs on the page. Example:

```
<html>
<head>
<script type="text/javascript">
<!--
window.onerror = function () {
   alert("An error occurred.");
}
//-->
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

To understand it in better way you can Try it yourself.

The **onerror** event handler provides three pieces of information to identify the exact nature of the error:

- **Error message** . The same message that the browser would display for the given error
- **URL** . The file in which the error occurred
- **Line number** . The line number in the given URL that caused the error

Here is the example to show how to extract this information

```
<html>
<head>
<script type="text/javascript">
<!--
window.onerror = function (msg, url, line) {
   alert("Message : " + msg );
   alert("url : " + url );
   alert("Line number : " + line );
}
//-->
```

```
</script>
</head>
<body>
<p>Click the following to see the result:</p>
<form>
<input type="button" value="Click Me" onclick="myFunc();" />
</form>
</body>
</html>
```

You can display extracted information in whatever way you think it is better.

To understand it in better way you can Try it yourself.

You can use **onerror** method to show an error message in case there is any problem in loading an image as follows:

```
<img src="myimage.gif"
    onerror="alert('An error occurred loading the image.')" />
```

You can use **onerror** with many HTML tags to display appropriate messages in case of errors.

JavaScript - Form Validation

Form validation used to occur at the server, after the client had entered all necessary data and then pressed the Submit button. If some of the data that had been entered by the client had been in the wrong form or was simply missing, the server would have to send all the data back to the client and request that the form be resubmitted with correct information. This was really a lengthy process and over burdening server.

JavaScript, provides a way to validate form's data on the client's computer before sending it to the web server. Form validation generally performs two functions.

- **Basic Validation** - First of all, the form must be checked to make sure data was entered into each form field that required it. This would need just loop through each field in the form and check for data.
- **Data Format Validation** - Secondly, the data that is entered must be checked for correct form and value. This would need to put more logic to test correctness of data.

We will take an example to understand the process of validation. Here is the simple form to proceed :

```
<html>
<head>
<title>Form Validation</title>
<script type="text/javascript">
<!--
// Form validation code will come here.
//-->
</script>
</head>
<body>
 <form action="/cgi-bin/test.cgi" name="myForm"
          onsubmit="return(validate());">
 <table cellspacing="2" cellpadding="2" border="1">
 <tr>
```

```
   <td align="right">Name</td>
   <td><input type="text" name="Name" /></td>
</tr>
<tr>
   <td align="right">EMail</td>
   <td><input type="text" name="EMail" /></td>
</tr>
<tr>
   <td align="right">Zip Code</td>
   <td><input type="text" name="Zip" /></td>
</tr>
<tr>
<td align="right">Country</td>
<td>
<select name="Country">
   <option value="-1" selected>[choose yours]</option>
   <option value="1">USA</option>
   <option value="2">UK</option>
   <option value="3">INDIA</option>
</select>
</td>
</tr>
<tr>
   <td align="right"></td>
   <td><input type="submit" value="Submit" /></td>
</tr>
</table>
</form>
</body>
</html>
```

Basic Form Validation:

First we will show how to do a basic form validation. In the above form we are calling **validate()** function to validate data when **onsubmit** event is occurring. Following is the implementation of this validate() function:

```
<script type="text/javascript">
<!--
// Form validation code will come here.
function validate()
{

   if( document.myForm.Name.value == "" )
   {
     alert( "Please provide your name!" );
     document.myForm.Name.focus() ;
     return false;
   }
   if( document.myForm.EMail.value == "" )
   {
     alert( "Please provide your Email!" );
     document.myForm.EMail.focus() ;
     return false;
   }
   if( document.myForm.Zip.value == "" ||
           isNaN( document.myForm.Zip.value ) ||
           document.myForm.Zip.value.length != 5 )
   {
```

```
      alert( "Please provide a zip in the format #####." );
      document.myForm.Zip.focus() ;
      return false;
   }
   if( document.myForm.Country.value == "-1" )
   {
      alert( "Please provide your country!" );
      return false;
   }
   return( true );
}
//-->
</script>
```

To understand it in better way you can Try it yourself.

Data Format Validation:

Now we will see how we can validate our entered form data before submitting it to the web server.

This example shows how to validate an entered email address which means email address must contain at least an @ sign and a dot (.). Also, the @ must not be the first character of the email address, and the last dot must at least be one character after the @ sign:

```
<script type="text/javascript">
<!--
function validateEmail()
{

   var emailID = document.myForm.EMail.value;
   atpos = emailID.indexOf("@");
   dotpos = emailID.lastIndexOf(".");
   if (atpos < 1 || ( dotpos - atpos < 2 ))
   {
       alert("Please enter correct email ID")
       document.myForm.EMail.focus() ;
       return false;
   }
   return( true );
}
//-->
</script>
```

To understand it in better way you can Try it yourself.

Javascript - Browsers Compatibility

It is important to understand the differences between different browsers in order to handle each in the way it is expected. So it is important to know which browser your Web page is running in.

To get information about the browser your Web page is currently running in, use the built-in **navigator** object.

Navigator Properties:

There are several Navigator related properties that you can use in your Web page. The following is a list of the names and descriptions of each:

Property	Description
appCodeName	This property is a string that contains the code name of the browser, *Netscape* for Netscape and *Microsoft Internet Explorer* for Internet Explorer.
appVersion	This property is a string that contains the version of the browser as well as other useful information such as its language and compatibility.
language	This property contains the two-letter abbreviation for the language that is used by the browser. Netscape only.
mimTypes[]	This property is an array that contains all MIME types supported by the client. Netscape only.
platform[]	This property is a string that contains the platform for which the browser was compiled."Win32" for 32-bit Windows operating systems
plugins[]	This property is an array containing all the plug-ins that have been installed on the client. Netscape only.
userAgent[]	This property is a string that contains the code name and version of the browser. This value is sent to the originating server to identify the client

Navigator Methods:

There are several Navigator-specific methods. Here is a list of their names and descriptions:

Method	Description
javaEnabled()	This method determines if JavaScript is enabled in the client. If JavaScript is enabled, this method returns true; otherwise, it returns false.
plugings.refresh	This method makes newly installed plug-ins available and populates the plugins array with all new plug-in names. Netscape only.
preference(name,value)	This method allows a signed script to get and set some Netscape preferences. If the second parameter is omitted, this method will return the value of the specified preference; otherwise, it sets the value. Netscape only.
taintEnabled()	This method returns true if data tainting is enabled and false otherwise.

Browser Detection:

There is a simple JavaScript which can be used to find out the name of a browser and then accordingly an HTML page can be served to the user.

```
<html>
<head>
<title>Browser Detection Example</title>
</head>
<body>
<script type="text/javascript">
<!--
var userAgent   = navigator.userAgent;
var opera       = (userAgent.indexOf('Opera') != -1);
var ie          = (userAgent.indexOf('MSIE') != -1);
var gecko       = (userAgent.indexOf('Gecko') != -1);
var netscape    = (userAgent.indexOf('Mozilla') != -1);
var version     = navigator.appVersion;

if (opera){
  document.write("Opera based browser");
  // Keep your opera specific URL here.
}else if (gecko){
  document.write("Mozilla based browser");
  // Keep your gecko specific URL here.
}else if (ie){
  document.write("IE based browser");
  // Keep your IE specific URL here.
}else if (netscape){
  document.write("Netscape based browser");
  // Keep your Netscape specific URL here.
}else{
  document.write("Unknown browser");
}
// You can include version to along with any above condition.
document.write("<br /> Browser version info : " + version );
//-->
</script>
</body>
</html>
```

To understand it in better way you can Try it yourself.

Javascript - The String Object

The **String** object let's you work with a series of characters and wraps Javascript's string primitive data type with a number of helper methods.

Because Javascript automatically converts between string primitives and String objects, you can call any of the helper methods of the String object on a string primitive.

Syntax:

Creating a **String** object:

```
var val = new String(string);
```

The *string* parameter is series of characters that has been properly encoded.

String Properties:

Here is a list of each property and their description.

Property	Description
constructor	Returns a reference to the String function that created the object.
length	Returns the length of the string.
prototype	The prototype property allows you to add properties and methods to an object.

String Methods

Here is a list of each method and its description.

Method	Description
charAt()	Returns the character at the specified index.
charCodeAt()	Returns a number indicating the Unicode value of the character at the given index.
concat()	Combines the text of two strings and returns a new string.
indexOf()	Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
lastIndexOf()	Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
localeCompare()	Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
match()	Used to match a regular expression against a string.
replace()	Used to find a match between a regular expression and a string, and to replace the matched substring with a new substring.
search()	Executes the search for a match between a regular expression and a specified string.

slice()	Extracts a section of a string and returns a new string.
split()	Splits a String object into an array of strings by separating the string into substrings.
substr()	Returns the characters in a string beginning at the specified location through the specified number of characters.
substring()	Returns the characters in a string between two indexes into the string.
toLocaleLowerCase()	The characters within a string are converted to lower case while respecting the current locale.
toLocaleUpperCase()	The characters within a string are converted to upper case while respecting the current locale.
toLowerCase()	Returns the calling string value converted to lower case.
toString()	Returns a string representing the specified object.
toUpperCase()	Returns the calling string value converted to uppercase.
valueOf()	Returns the primitive value of the specified object.

String HTML wrappers

Here is a list of each method which returns a copy of the string wrapped inside the appropriate HTML tag.

Method	Description
anchor()	Creates an HTML anchor that is used as a hypertext target.
big()	Creates a string to be displayed in a big font as if it were in a <big> tag.
blink()	Creates a string to blink as if it were in a <blink> tag.
bold()	Creates a string to be displayed as bold as if it were in a <b> tag.
fixed()	Causes a string to be displayed in fixed-pitch font as if it were in a <tt> tag
fontcolor()	Causes a string to be displayed in the specified color as if it were in a <font color="color"> tag.

fontsize()	Causes a string to be displayed in the specified font size as if it were in a <font size="size"> tag.
italics()	Causes a string to be italic, as if it were in an <i> tag.
link()	Creates an HTML hypertext link that requests another URL.
small()	Causes a string to be displayed in a small font, as if it were in a <small> tag.
strike()	Causes a string to be displayed as struck-out text, as if it were in a <strike> tag.
sub()	Causes a string to be displayed as a subscript, as if it were in a <sub> tag
sup()	Causes a string to be displayed as a superscript, as if it were in a <sup> tag

Javascript - The Arrays Object

The **Array** object let's you store multiple values in a single variable.

Syntax:

Creating a **Array** object:

```
var fruits = new Array( "apple", "orange", "mango" );
```

The *Array* parameter is a list of strings or integers. When you specify a single numeric parameter with the Array constructor, you specify the initial length of the array. The maximum length allowed for an array is 4,294,967,295.

You can create array by simply assigning values as follows:

```
var fruits = [ "apple", "orange", "mango" ];
```

You will use ordinal numbers to access and to set values inside an array as follows:

- fruits[0] is the first element
- fruits[1] is the second element
- fruits[2] is the third element

Array Properties:

Here is a list of each property and their description.

Property	Description
constructor	Returns a reference to the array function that created the object.
index	The property represents the zero-based index of the match in the string
input	This property is only present in arrays created by regular expression matches.
length	Reflects the number of elements in an array.
prototype	The prototype property allows you to add properties and methods to an object.

Array Methods

Here is a list of each method and its description.

Method	Description
concat()	Returns a new array comprised of this array joined with other array(s) and/or value(s).
every()	Returns true if every element in this array satisfies the provided testing function.
filter()	Creates a new array with all of the elements of this array for which the provided filtering function returns true.
forEach()	Calls a function for each element in the array.
indexOf()	Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found.
join()	Joins all elements of an array into a string.
lastIndexOf()	Returns the last (greatest) index of an element within the array equal to the specified value, or -1 if none is found.
map()	Creates a new array with the results of calling a provided function on every element in this array.
pop()	Removes the last element from an array and returns that element.

push()	Adds one or more elements to the end of an array and returns the new length of the array.
reduce()	Apply a function simultaneously against two values of the array (from left-to-right) as to reduce it to a single value.
reduceRight()	Apply a function simultaneously against two values of the array (from right-to-left) as to reduce it to a single value.
reverse()	Reverses the order of the elements of an array -- the first becomes the last, and the last becomes the first.
shift()	Removes the first element from an array and returns that element.
slice()	Extracts a section of an array and returns a new array.
some()	Returns true if at least one element in this array satisfies the provided testing function.
toSource()	Represents the source code of an object
sort()	Sorts the elements of an array.
splice()	Adds and/or removes elements from an array.
toString()	Returns a string representing the array and its elements.
unshift()	Adds one or more elements to the front of an array and returns the new length of the array.

JavaScript - The Date Object

The Date object is a datatype built into the JavaScript language. Date objects are created with the **new Date( )** as shown below.

Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time.

The ECMAScript standard requires the Date object to be able to represent any date and time, to millisecond precision, within 100 million days before or after 1/1/1970. This is a range of plus or minus 273,785 years, so the JavaScript is able to represent date and time till year 275755.

Syntax:

Here are different variant of Date() constructor:

```
new Date( )
new Date(milliseconds)
new Date(datestring)
new Date(year,month,date[,hour,minute,second,millisecond ])
```

Note: Paramters in the brackets are always optional

Here is the description of the parameters:

- **No Argument:** With no arguments, the Date( ) constructor creates a Date object set to the current date and time.
- **milliseconds:** When one numeric argument is passed, it is taken as the internal numeric representation of the date in milliseconds, as returned by the getTime( ) method. For example, passing the argument 5000 creates a date that represents five seconds past midnight on 1/1/70.
- **datestring:**When one string argument is passed, it is a string representation of a date, in the format accepted by the Date.parse( ) method.
- **7 agruments:** To use the last form of constructor given above, Here is the description of each argument:
    1. **year:** Integer value representing the year. For compatibility (in order to avoid the Y2K problem), you should always specify the year in full; use 1998, rather than 98.
    2. **month:** Integer value representing the month, beginning with 0 for January to 11 for December.
    3. **date:** Integer value representing the day of the month.
    4. **hour:** Integer value representing the hour of the day (24-hour scale).
    5. **minute:** Integer value representing the minute segment of a time reading.
    6. **second:** Integer value representing the second segment of a time reading.
    7. **millisecond:** Integer value representing the millisecond segment of a time reading.

Date Properties:

Here is a list of each property and their description.

Property	Description
constructor	Specifies the function that creates an object's prototype.
prototype	The prototype property allows you to add properties and methods to an object.

Date Methods:

Here is a list of each method and its description.

Method	Description
Date()	Returns today's date and time
getDate()	Returns the day of the month for the specified date according to

	local time.
getDay()	Returns the day of the week for the specified date according to local time.
getFullYear()	Returns the year of the specified date according to local time.
getHours()	Returns the hour in the specified date according to local time.
getMilliseconds()	Returns the milliseconds in the specified date according to local time.
getMinutes()	Returns the minutes in the specified date according to local time.
getMonth()	Returns the month in the specified date according to local time.
getSeconds()	Returns the seconds in the specified date according to local time.
getTime()	Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
getTimezoneOffset()	Returns the time-zone offset in minutes for the current locale.
getUTCDate()	Returns the day (date) of the month in the specified date according to universal time.
getUTCDay()	Returns the day of the week in the specified date according to universal time.
getUTCFullYear()	Returns the year in the specified date according to universal time.
getUTCHours()	Returns the hours in the specified date according to universal time.
getUTCMilliseconds()	Returns the milliseconds in the specified date according to universal time.
getUTCMinutes()	Returns the minutes in the specified date according to universal time.
getUTCMonth()	Returns the month in the specified date according to universal time.
getUTCSeconds()	Returns the seconds in the specified date according to universal time.

getYear()	**Deprecated** - Returns the year in the specified date according to local time. Use getFullYear instead.
setDate()	Sets the day of the month for a specified date according to local time.
setFullYear()	Sets the full year for a specified date according to local time.
setHours()	Sets the hours for a specified date according to local time.
setMilliseconds()	Sets the milliseconds for a specified date according to local time.
setMinutes()	Sets the minutes for a specified date according to local time.
setMonth()	Sets the month for a specified date according to local time.
setSeconds()	Sets the seconds for a specified date according to local time.
setTime()	Sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.
setUTCDate()	Sets the day of the month for a specified date according to universal time.
setUTCFullYear()	Sets the full year for a specified date according to universal time.
setUTCHours()	Sets the hour for a specified date according to universal time.
setUTCMilliseconds()	Sets the milliseconds for a specified date according to universal time.
setUTCMinutes()	Sets the minutes for a specified date according to universal time.
setUTCMonth()	Sets the month for a specified date according to universal time.
setUTCSeconds()	Sets the seconds for a specified date according to universal time.
setYear()	**Deprecated -** Sets the year for a specified date according to local time. Use setFullYear instead.
toDateString()	Returns the "date" portion of the Date as a human-readable string.
toGMTString()	**Deprecated -** Converts a date to a string, using the Internet GMT conventions. Use toUTCString instead.

toLocaleDateString()	Returns the "date" portion of the Date as a string, using the current locale's conventions.
toLocaleFormat()	Converts a date to a string, using a format string.
toLocaleString()	Converts a date to a string, using the current locale's conventions.
toLocaleTimeString()	Returns the "time" portion of the Date as a string, using the current locale's conventions.
toSource()	Returns a string representing the source for an equivalent Date object; you can use this value to create a new object.
toString()	Returns a string representing the specified Date object.
toTimeString()	Returns the "time" portion of the Date as a human-readable string.
toUTCString()	Converts a date to a string, using the universal time convention.
valueOf()	Returns the primitive value of a Date object.

Date Static Methods:

In addition to the many instance methods listed previously, the Date object also defines two static methods. These methods are invoked through the Date( ) constructor itself:

Method	Description
Date.parse( )	Parses a string representation of a date and time and returns the internal millisecond representation of that date.
Date.UTC( )	Returns the millisecond representation of the specified UTC date and time.

Javascript - The Math Object

The **math** object provides you properties and methods for mathematical constants and functions.

Unlike the other global objects, *Math* is not a constructor. All properties and methods of Math are static and can be called by using *Math* as an object without creating it.

Thus, you refer to the constant pi as **Math.PI** and you call the *sine* function as **Math.sin(x)**, where x is the method's argument.

Syntax:

Here is the simple syntax to call properties and methods of Math.

```
var pi_val = Math.PI;
var sine_val = Math.sin(30);
```

Math Properties:

Here is a list of each property and their description.

Property	Description
E	Euler's constant and the base of natural logarithms, approximately 2.718.
LN2	Natural logarithm of 2, approximately 0.693.
LN10	Natural logarithm of 10, approximately 2.302.
LOG2E	Base 2 logarithm of E, approximately 1.442.
LOG10E	Base 10 logarithm of E, approximately 0.434.
PI	Ratio of the circumference of a circle to its diameter, approximately 3.14159.
SQRT1_2	Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
SQRT2	Square root of 2, approximately 1.414.

Math Methods

Here is a list of each method and its description.

Method	Description
abs()	Returns the absolute value of a number.
acos()	Returns the arccosine (in radians) of a number.
asin()	Returns the arcsine (in radians) of a number.
atan()	Returns the arctangent (in radians) of a number.
atan2()	Returns the arctangent of the quotient of its arguments.

ceil()	Returns the smallest integer greater than or equal to a number.
cos()	Returns the cosine of a number.
exp()	Returns E^N, where N is the argument, and E is Euler's constant, the base of the natural logarithm.
floor()	Returns the largest integer less than or equal to a number.
log()	Returns the natural logarithm (base E) of a number.
max()	Returns the largest of zero or more numbers.
min()	Returns the smallest of zero or more numbers.
pow()	Returns base to the exponent power, that is, base exponent.
random()	Returns a pseudo-random number between 0 and 1.
round()	Returns the value of a number rounded to the nearest integer.
sin()	Returns the sine of a number.
sqrt()	Returns the square root of a number.
tan()	Returns the tangent of a number.
toSource()	Returns the string "Math".

Regular Expressions and RegExp Object

A regular expression is an object that describes a pattern of characters.

The JavaScript **RegExp** class represents regular expressions, and both String and **RegExp** define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.

Syntax:

A regular expression could be defined with the RegExp( ) constructor like this:

```
var pattern = new RegExp(pattern, attributes);

or simply

var pattern = /pattern/attributes;
```

Here is the description of the parameters:

- **pattern:** A string that specifies the pattern of the regular expression or another regular expression.
- **attributes:** An optional string containing any of the "g", "i", and "m" attributes that specify global, case-insensitive, and multiline matches, respectively.

Brackets:

Brackets ([]) have a special meaning when used in the context of regular expressions. They are used to find a range of characters.

Expression	Description
[...]	Any one character between the brackets.
[^...]	Any one character not between the brackets.
[0-9]	It matches any decimal digit from 0 through 9.
[a-z]	It matches any character from lowercase a through lowercase z.
[A-Z]	It matches any character from uppercase A through uppercase Z.
[a-Z]	It matches any character from lowercase a through uppercase Z.

The ranges shown above are general; you could also use the range [0-3] to match any decimal digit ranging from 0 through 3, or the range [b-v] to match any lowercase character ranging from b through v.

Quantifiers:

The frequency or position of bracketed character sequences and single characters can be denoted by a special character. Each pecial character having a specific connotation. The +, *, ?, and $ flags all follow a character sequence.

Expression	Description
p+	It matches any string containing at least one p.
p*	It matches any string containing zero or more p's.
p?	It matches any string containing one or more p's.
p{**N**}	It matches any string containing a sequence of **N** p's
p{2,3}	It matches any string containing a sequence of two or three p's.

p{2, }	It matches any string containing a sequence of at least two p's.
p$	It matches any string with p at the end of it.
^p	It matches any string with p at the beginning of it.

Examples:

Following examples will clear your concepts about matching chracters.

Expression	Description
[^a-zA-Z]	It matches any string not containing any of the characters ranging from a through z and A through Z.
p.p	It matches any string containing p, followed by any character, in turn followed by another p.
^.{2}$	It matches any string containing exactly two characters.
(.*)	It matches any string enclosed within and .
p(hp)*	It matches any string containing a p followed by zero or more instances of the sequence hp.

Literal characters:

Character	Description
Alphanumeric	Itself
\0	The NUL character (\u0000)
\t	Tab (\u0009)
\n	Newline (\u000A)
\v	Vertical tab (\u000B)
\f	Form feed (\u000C)
\r	Carriage return (\u000D)

\xnn	The Latin character specified by the hexadecimal number nn; for example, \x0A is the same as \n
\uxxxx	The Unicode character specified by the hexadecimal number xxxx; for example, \u0009 is the same as \t
\cX	The control character ^X; for example, \cJ is equivalent to the newline character \n

Metacharacters

A metacharacter is simply an alphabetical character preceded by a backslash that acts to give the combination a special meaning.

For instance, you can search for large money sums using the '\d' metacharacter: **/([\d]+)000/**, Here **\d** will search for any string of numerical character.

Following is the list of metacharacters which can be used in PERL Style Regular Expressions.

```
Character               Description
.               a single character
\s              a whitespace character (space, tab, newline)
\S              non-whitespace character
\d              a digit (0-9)
\D              a non-digit
\w              a word character (a-z, A-Z, 0-9, _)
\W              a non-word character
[\b]            a literal backspace (special case).
[aeiou]         matches a single character in the given set
[^aeiou]        matches a single character outside the given set
(foo|bar|baz)   matches any of the alternatives specified
```

Modifiers

Several modifiers are available that can make your work with regexps much easier, like case sensitivity, searching in multiple lines etc.

Modifier	Description
i	Perform case-insensitive matching.
m	Specifies that if the string has newline or carriage return characters, the ^ and $ operators will now match against a newline boundary, instead of a string boundary
g	Perform a global matchthat is, find all matches rather than stopping after the first match.

RegExp Properties:

Here is a list of each property and their description.

Property	Description
constructor	Specifies the function that creates an object's prototype.
global	Specifies if the "g" modifier is set.
ignoreCase	Specifies if the "i" modifier is set.
lastIndex	The index at which to start the next match.
multiline	Specifies if the "m" modifier is set.
source	The text of the pattern.

RegExp Methods:

Here is a list of each method and its description.

Method	Description
exec()	Executes a search for a match in its string parameter.
test()	Tests for a match in its string parameter.
toSource()	Returns an object literal representing the specified object; you can use this value to create a new object.
toString()	Returns a string representing the specified object.

Further Detail:

Refer to the link http://www.tutorialspoint.com/javascript

▪ Learn JDBC ▪ Java Examples ▪ Learn Best Practices ▪ Learn Python ▪ Learn Ruby ▪ Learn Ruby on Rails ▪ Learn SQL ▪ Learn MySQL ▪ Learn AJAX ▪ Learn C Programming ▪ Learn C++ Programming ▪ Learn CGI with PERL ▪ Learn DLL ▪ Learn ebXML ▪ Learn Euphoria ▪ Learn GDB Debugger ▪ Learn Makefile ▪ Learn Parrot ▪ Learn Perl Script ▪ Learn PHP Script ▪ Learn Six Sigma ▪ Learn SEI CMMI ▪ Learn WiMAX ▪ Learn Telecom Billing	▪ Learn HTTP ▪ Learn JavaScript ▪ Learn jQuery ▪ Learn Prototype ▪ Learn script.aculo.us ▪ Web Developer's Guide ▪ Learn RADIUS ▪ Learn RSS ▪ Learn SEO Techniques ▪ Learn SOAP ▪ Learn UDDI ▪ Learn Unix Sockets ▪ Learn Web Services ▪ Learn XML-RPC ▪ Learn UML ▪ Learn UNIX ▪ Learn WSDL ▪ Learn i-Mode ▪ Learn GPRS ▪ Learn GSM ▪ Learn WAP ▪ Learn WML ▪ Learn Wi-Fi
webmaster@TutorialsPoint.com	