Chapter – 3

Scripting Language

Introduction to Scripting Language:

A scripting language is a programming language designed for integrating and communicating with other programming languages. Some of the most widely used scripting languages are JavaScript, VBScript, PHP, Perl, Python, Ruby, ASP and Tcl. Since a scripting language is normally used in conjunction with another programming language, they are often found alongside HTML, Java or C++.

One common distinction between a scripting language and a language used for writing entire applications is that, while a programming language is typically compiled first before being allowed to run, scripting languages are interpreted from source code or bytecode one command at a time.

Although scripts are widely employed in the programming world, they have recently become more associated with the World Wide Web, where they have been used extensively to create dynamic Web pages. While technically there are many client-side scripting languages that can be used on the Web, in practice it means using JavaScript.

Differentiate between client side and server side scripting language:

Client Side Scripting Language	Server Side Scripting Language
Client side scripting is used when the user's browser already has all the code and the page is altered on the basis of the users input.	Server side scripting is used to create dynamic pages based a number of conditions when the users browser makes a request to the server.
The Web Browser executes the client side scripting that resides at the user's computer.	The Web Server executes the server side scripting that produces the page to be sent to the browser.
The browser receives the page sent by the server and executes the client-side scripts.	Server executes server-side scripts to send out a page but it does not execute client-side scripts.
Client side scripting cannot be used to connect to the databases on the web server.	Server side scripting is used to connect to the databases that reside on the web server.
Client side scripting can't access the file system that resides at the web server.	Server side scripting can access the file system residing at the web server.
The files and settings that are local at the user's computer can be accessed using Client side scripting.	The settings that belong to Web server can be accessed using Server side scripting.
Client side scripting is possible to be blocked by the user.	Server side scripting can't be blocked by the user.
Response from a client-side script is faster as compared to a server-side script	Response from a server-side script is slower as compared to a client-side script

because the scripts are processed on the local computer.	because the scripts are processed on the remote computer.
Examples of Client side scripting languages : JavaScript, VB script, etc.	Examples of Server side scripting languages: PHP, JSP, ASP, ASP.Net, Ruby, Perl and many more.

JAVASCRIPT:

JavaScript is a lightweight, interpreted programming language. It is designed for creating network-centric applications. It is complimentary to and integrated with Java. JavaScript is very easy to implement because it is integrated with HTML. It is open and cross-platform.

JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. It is an interpreted programming language with object-oriented capabilities.

JavaScript was first known as **LiveScript,** but Netscape changed its name to JavaScript, possibly because of the excitement being generated by Java. JavaScript made its first appearance in Netscape 2.0 in 1995 with the name **LiveScript**. The general-purpose core of the language has been embedded in Netscape, Internet Explorer, and other web browsers.

FEATURES OF JAVASCRIPT:

- 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 it works.

CLIENT-SIDE JAVASCRIPT:

Client-side JavaScript is the most common form of the language. The script should be included in or referenced by an HTML document for the code to be interpreted by the browser. It means that a web page need not be a static HTML, but can include programs that interact with the user, control the browser, and dynamically create HTML content.

The JavaScript client-side mechanism provides many advantages over traditional CGI server-side scripts. For example, we might use JavaScript to check if the user has entered a valid e-mail address in a form field.

The JavaScript code is executed when the user submits the form, and only if all the entries are valid, they would be submitted to the Web Server.

JavaScript can be used to trap user-initiated events such as button clicks, link navigation, and other actions that the user initiates explicitly or implicitly.

ADVANTAGES OF JAVASCRIPT:

The merits of using JavaScript are:

- **Less Server Interaction**:

We can validate user input before sending the page off to the server. This saves server traffic, which means less load on our server.

- **Immediate Feedback To The Visitors**:

They don't have to wait for a page reload to see if they have forgotten to enter something.

- **Increased interactivity**:

We can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.

- **Richer interfaces**:

We can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

LIMITATIONS OF JAVASCRIPT:

We cannot treat JavaScript as a full-fledged programming language. It lacks the following important features:

- Client-side JavaScript does not allow the reading or writing of files. This has been kept for security reason.
- JavaScript cannot be used for networking applications because there is no such support available.
- JavaScript doesn't have any multithreading or multiprocessor capabilities.

JAVASCRIPT SYNTAX:

JavaScript can be implemented using JavaScript statements that are placed within the **<script>... </script>** HTML tags in a web page. We can place the **<script>** tags, containing our JavaScript, anywhere within our web page, but it is normally recommended that we should keep it within the **<head>** tags.

The <script> tag alerts the browser program to start interpreting all the text between these tags as a script. A simple syntax of our JavaScript will appear as follows.

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

The script tag takes two important attributes:

- **Language**:

This attribute specifies what scripting language we 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 JavaScript segment will look like:

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

Example:

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

1. Whitespace and Line Breaks:

JavaScript ignores spaces, tabs, and newlines that appear in JavaScript programs. We can use spaces, tabs, and newlines freely in our program and we are free to format and indent our programs in a neat and consistent way that makes the code easy to read and understand.

2. 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 us to omit this semicolon if each of our statements are 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, we must use semicolons:

```
<script language="JavaScript" type="text/JavaScript">
    var1 = 10; var2 = 20;
</script>
```

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

3. Case Sensitivity:

JavaScript is a case-sensitive language. This means that the language keywords, variables, function names, and any other identifiers must always be typed with a consistent capitalization of letters. So the identifiers **Time** and **TIME** will convey different meanings in JavaScript.

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

4. Comments in JavaScript:

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

- 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 //-->.

Example

The following example shows how to use comments in JavaScript.

```
<script language="javascript" type="text/javascript">
  <!--
    // This is a comment. It is similar to comments in C++
    /*
    * This is a multiline comment in JavaScript
    * It is very similar to comments in C Programming
    */
  //-->
</script>
```

Using JavaScript in HTML Document:

Example 1:

```
<!DOCTYPE html>
<html>
<body>
    <h1>My First JavaScript</h1>
    <button type="button" onclick="document.getElementById('demo').innerHTML = Date()">
    Click me to display Date and Time. </button>
    <p id="demo"></p>
</body>
```

```
</html>
```

Example 2: JavaScript in Head Section

```
<html>
  <head>
    <script type="text/javascript">
        function sayHello() {
          alert("Hello World")
        }
    </script>
      </head>
    <body>
    <input type="button" onclick="sayHello()" value="Say Hello" />
  </body>
</html>
```

Example 3: JavaScript in External File

```
function sayHello() {
          alert("Hello World")
}
```

Save this file with name hello.js then after,

```
<html>
  <head>
    <script type="text/javascript" src = "hello.js"> </script>
   </head>
   <body>
    <input type="button" onclick="sayHello()" value="Say Hello" />
  </body>
</html>
```

PROGRAMMING FUNDAMENTALS:

1. JAVASCRIPT DATA TYPES:

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

JavaScript allows us 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. In addition to these primitive data types, JavaScript supports a composite data type known as **object**.

Note: JavaScript does not make a distinction between integer values and floating-point values. All numbers in JavaScript are represented as floating-point values. JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.

2. JavaScript Variables:

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

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

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

We can also declare multiple variables with the same **var** keyword as follows:

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

Storing a value in a variable is called **variable initialization**. We can do variable initialization at the time of variable creation or at a later point in time when we need that variable.

For instance, we might create a variable named **money** and assign the value 2000.50 to it later. For another variable, we can assign a value at the time of initialization as follows.

```
<script type="text/javascript">
    var name = "Ali";
    var money;
    money = 2000.50;
</script>
```

Note: Use the **var** keyword only for declaration or initialization, once for the life of any variable name in a document. We should not re-declare same variable twice.

JavaScript is **untyped** language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, we don't have to tell JavaScript during variable declaration what type of value the variable will hold. The value type of a variable can change during the execution of a program and JavaScript takes care of it automatically.

3. JavaScript Variable Scope:

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

- **Global Variables**: A global variable has global scope which means it can be defined anywhere in our 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.

Within the body of a function, a local variable takes precedence over a global variable with the same name. If we declare a local variable or function parameter with the same name as a global variable, we effectively hide the global variable. Take a look into the following example.

```
<html>
  <body onload = checkscope();>
    <script type = "text/javascript">
        var myVar = "global"; // Declare a global variable
        function checkscope( ) {
          var myVar = "local";  // Declare a local variable
          document.write(myVar);
        }
    </script>
  </body>
</html>
```

4. JavaScript Variable Names:

While naming variables in JavaScript, keep the following rules in mind.

- We should not use any of the JavaScript reserved keywords as a 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 an 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.

5. JavaScript Reserved Words:

A list of all the reserved words in JavaScript are given in the following table. They cannot be used as JavaScript variables, functions, methods, loop labels, or any object names.

abstract	class	enum	function	long
boolean	const	export	goto	native
break	continue	extends	if	new
byte	debugger	false	implements	null
case	default	final	import	package
catch	class	finally	instanceof	private
char	do	float	int	protected
delete	else	for	interface	Public
return	short	static	typeof	Var

void	while	super	switch	this
volatile	with	in	synchronized	Throw
throws	transient	true	try	double

6. Operators:

Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**. JavaScript supports the following types of operators.

a. Arithmetic Operators:

JavaScript supports the following arithmetic operators. Assume variable A holds 10 and variable B holds 20, then:

S.N.	Operator and Description
1	**+ (Addition)** Adds two operands **Ex:** A + B will give 30
2	**- (Subtraction)** Subtracts the second operand from the first **Ex:** A - B will give -10
3	*** (Multiplication)** Multiply both operands **Ex:** A * B will give 200
4	**/ (Division)** Divide the numerator by the denominator **Ex:** B / A will give 2
5	**% (Modulus)** Outputs the remainder of an integer division **Ex:** B % A will give 0
6	**++ (Increment)** Increases an integer value by one **Ex:** A++ will give 11
7	**-- (Decrement)** Decreases an integer value by one **Ex:** A-- will give 9

Note: Addition operator (+) works for Numeric as well as Strings. e.g. "a" + 10 will give "a10".

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = 33;
        var b = 10;
        var c = "Test";
        var linebreak = "<br />";
```

```
        document.write("a + b = ");
        result = a + b;
        document.write(result);
        document.write(linebreak);
        document.write("a - b = ");
        result = a - b;
        document.write(result);
        document.write(linebreak);

        document.write("a / b = ");
        result = a / b;
        document.write(result);
        document.write(linebreak);

        document.write("a % b = ");
        result = a % b;
        document.write(result);
        document.write(linebreak);

        document.write("a + b + c = ");
        result = a + b + c;
        document.write(result);
        document.write(linebreak);

        a = ++a;
        document.write("++a = ");
        result = ++a;
        document.write(result);
        document.write(linebreak);

        b = --b;
        document.write("--b = ");
        result = --b;
        document.write(result);
        document.write(linebreak);
    </script>
  </body>
</html>
```

Output

```
a + b = 43
a - b = 23
a / b = 3.3
a % b = 3
a + b + c = 43Test
++a = 35
--b = 8
```

b. <u>Comparison Operators:</u>

JavaScript supports the following comparison operators. Assume variable A holds 10 and variable B holds 20, then:

S.No.	Operator and Description
1	= = **(Equal)**

	Checks if the value of two operands are equal or not, if yes, then the condition becomes true. **Ex:** (A == B) is not true.
2	**!= (Not Equal)** Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true. **Ex:** (A != B) is true.
3	**> (Greater than)** Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A > B) is not true.
4	**< (Less than)** Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true. **Ex:** (A < B) is true.
5	**>= (Greater than or Equal to)** Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A >= B) is not true.
6	**<= (Less than or Equal to)** Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true. **Ex:** (A <= B) is true.

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = 10;
        var b = 20;
        var linebreak = "<br />";

        document.write("(a == b) => ");
        result = (a == b);
        document.write(result);
        document.write(linebreak);

        document.write("(a < b) => ");
        result = (a < b);
        document.write(result);
        document.write(linebreak);

        document.write("(a > b) => ");
        result = (a > b);
        document.write(result);
```

Output

```
(a == b) => false
(a < b) => true
(a > b) => false
(a != b) => true
(a >= b) => false
a <= b) => true
```

```
        document.write(linebreak);

        document.write("(a != b) => ");
        result = (a != b);
        document.write(result);
        document.write(linebreak);

        document.write("(a >= b) => ");
        result = (a >= b);
        document.write(result);
        document.write(linebreak);

        document.write("(a <= b) => ");
        result = (a <= b);
        document.write(result);
        document.write(linebreak);
    </script>
  </body>
</html>
```

c. Logical Operators:

JavaScript supports the following logical operators. Assume variable A holds 10 and variable B holds 20, then:

S.No.	Operator and Description
1	**&& (Logical AND)** If both the operands are non-zero, then the condition becomes true. **Ex:** (A && B) is true.
2	**|| (Logical OR)** If any of the two operands are non-zero, then the condition becomes true. **Ex:** (A || B) is true.
3	**! (Logical NOT)** Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false. **Ex:** ! (A && B) is false.

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = true;
        var b = false;
        var linebreak = "<br />";

        document.write("(a && b) => ");
```

Output

```
(a && b) => false
(a || b) => true
!(a && b) => true
```

```
            result = (a && b);
            document.write(result);
            document.write(linebreak);

            document.write("(a || b) => ");
            result = (a || b);
            document.write(result);
            document.write(linebreak);

            document.write("!(a && b) => ");
            result = (!(a && b));
            document.write(result);
            document.write(linebreak);
        </script>
    </body>
</html>
```

d. Bitwise Operators:

JavaScript supports the following bitwise operators. Assume variable A holds 2 and variable B holds 3, then:

S.No.	Operator and Description
1	**& (Bitwise AND)** It performs a Boolean AND operation on each bit of its integer arguments. **Ex:** (A & B) is 2.
2	**| (BitWise OR)** It performs a Boolean OR operation on each bit of its integer arguments. **Ex:** (A | B) is 3.
3	**^ (Bitwise XOR)** 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. **Ex:** (A ^ B) is 1.
4	**~ (Bitwise Not)** It is a unary operator and operates by reversing all the bits in the operand. **Ex:** (~B) is -4.
5	**<< (Left Shift)** It moves all the 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 it by 2, shifting two positions is equivalent to multiplying by 4, and so on. **Ex:** (A << 1) is 4.
6	**>> (Right Shift)** Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand. **Ex:** (A >> 1) is 1.
7	**>>> (Right shift with Zero)**

	This operator is just like the >> operator, except that the bits shifted in on the left are always zero. **Ex:** (A >>> 1) is 1.

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = 2; // Bit presentation 10
        var b = 3; // Bit presentation 11
        var linebreak = "<br />";

        document.write("(a & b) => ");
        result = (a & b);
        document.write(result);
        document.write(linebreak);

        document.write("(a | b) => ");
        result = (a | b);
        document.write(result);
        document.write(linebreak);

        document.write("(a ^ b) => ");
        result = (a ^ b);
        document.write(result);
        document.write(linebreak);

        document.write("(~b) => ");
        result = (~b);
        document.write(result);
        document.write(linebreak);

        document.write("(a << b) => ");
        result = (a << b);
        document.write(result);
        document.write(linebreak);

        document.write("(a >> b) => ");
        result = (a >> b);
        document.write(result);
        document.write(linebreak);
    </script>
  </body>
</html>
```

```
(a & b) => 2
(a | b) => 3
(a ^ b) => 1
(~b) => -4
(a << b) => 16
(a >> b) => 0
```

e. Assignment Operators:

JavaScript supports the following assignment operators:

S.No.	Operator and Description
1	**= (Simple Assignment)** Assigns values from the right side operand to the left side operand **Ex:** C = A + B will assign the value of A + B into C
2	**+= (Add and Assignment)** It adds the right operand to the left operand and assigns the result to the left operand. **Ex:** C += A is equivalent to C = C + A
3	**−= (Subtract and Assignment)** It subtracts the right operand from the left operand and assigns the result to the left operand. **Ex:** C -= A is equivalent to C = C - A
4	***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand. **Ex:** C *= A is equivalent to C = C * A
5	**/= (Divide and Assignment)** It divides the left operand with the right operand and assigns the result to the left operand. **Ex:** C /= A is equivalent to C = C / A
6	**%= (Modules and Assignment)** It takes modulus using two operands and assigns the result to the left operand. **Ex:** C %= A is equivalent to C = C % A

Note: Same logic applies to Bitwise operators so they will become like <<=, >>=, >>=, &=, |= and ^=.

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = 33;
        var b = 10;
        var linebreak = "<br />";

        document.write("Value of a => (a = b) => ");
        result = (a = b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a += b) => ");
        result = (a += b);
        document.write(result);
        document.write(linebreak);
```

Output

```
Value of a => (a = b) => 10
Value of a => (a += b) => 20
Value of a => (a -= b) => 10
Value of a => (a *= b) => 100
Value of a => (a /= b) => 10
Value of a => (a %= b) => 0
```

```
        document.write("Value of a => (a -= b) => ");
        result = (a -= b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a *= b) => ");
        result = (a *= b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a /= b) => ");
        result = (a /= b);
        document.write(result);
        document.write(linebreak);

        document.write("Value of a => (a %= b) => ");
        result = (a %= b);
        document.write(result);
        document.write(linebreak);
    </script>
  </body>
</html>
```

f. Conditional Operator (? :):

The conditional operator first evaluates an expression for a true or false value and then executes one of the two given statements depending upon the result of the evaluation.

S.No.	Operator and Description
1	**? : (Conditional)** If Condition is true? Then value X : Otherwise value Y

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = 10;
        var b = 20;
        var linebreak = "<br />";

        document.write ("((a > b) ? 100 : 200) => ");
        result = (a > b) ? 100 : 200;
        document.write(result);
        document.write(linebreak);

        document.write ("((a < b) ? 100 : 200) => ");
```

Output

```
((a > b) ? 100 : 200) => 200
((a < b) ? 100 : 200) => 100
```

```
        result = (a < b) ? 100 : 200;
        document.write(result);
        document.write(linebreak);
    </script>
  </body>
</html>
```

g. typeof Operator:

The **typeof** operator 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.

Here is a list of the return values for the **typeof** Operator.

Type	String Returned by typeof
Number	"number"
String	"string"
Boolean	"boolean"
Object	"object"
Function	"function"
Undefined	"undefined"
Null	"object"

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var a = 10;
        var b = "String";
        var linebreak = "<br />";

        result = (typeof b == "string" ? "B is String" : "B is Numeric");
        document.write("Result => ");
        document.write(result);
        document.write(linebreak);

        result = (typeof a == "string" ? "A is String" : "A is Numeric");
        document.write("Result => ");
        document.write(result);
        document.write(linebreak);
    </script>
  </body>
</html>
```

Output

```
Result => B is String
Result => A is Numeric
```

7. Control Flow Statements:

While writing a program, there may be a situation when you need to adopt one out of a given set of paths. In such cases, you need to use conditional statements that allow our program to make correct decisions and perform right actions.

JavaScript supports conditional statements which are used to perform different actions based on different conditions.

a. 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
}
```

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var age = 20;
        if( age > 18 ){
          document.write("<b>Qualifies for driving</b>");
        }
    </script>
  </body>
</html>
```

b. if...else statement:

The **'if...else'** statement is the next form of control statement that allows JavaScript to execute statements in a 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
}
```

Example:

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

```
        var age = 15;
        if( age > 18 ){
          document.write("<b>Qualifies for driving</b>");
        }
        else{
          document.write("<b>Does not qualify for driving</b>");
        }
    </script>
  </body>
</html>
```

c. if...else if... statement:

The **if...else if...** statement is an advanced form of **if...else** that allows JavaScript to make a 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
}
```

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var book = "maths";
        if( book == "history" ){
          document.write("<b>History Book</b>");
        }
        else if( book == "maths" ){
          document.write("<b>Maths Book</b>");
        }
        else if( book == "economics" ){
          document.write("<b>Economics Book</b>");
        }
        else{
          document.write("<b>Unknown Book</b>");
        }
```

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

d. Switch Statement:

The objective of a switch statement is to give an expression to evaluate 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.

Syntax:

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

Example:

```
<html>
  <body>
      <script type="text/javascript">
        var grade='A';
        document.write("Entering switch block<br />");
        switch (grade)
        {
          case 'A': document.write("Good job<br />");
          break;

          case 'B': document.write("Pretty good<br />");
          break;

          case 'C': document.write("Passed<br />");
          break;

          case 'D': document.write("Not so good<br />");
          break;

          case 'F': document.write("Failed<br />");
          break;
```

```
            default:  document.write("Unknown grade<br />")
         }
         document.write("Exiting switch block");
      </script>
   </body>
</html>
```

8. Loops:

While writing a program, we may encounter a situation where we need to perform an action over and over again. In such situations, we would need to write loop statements to reduce the number of lines. JavaScript supports all the necessary loops to ease down the pressure of programming.

a. The while Loop:

The most basic loop in JavaScript is the **while** loop which would be discussed in this chapter. The purpose of a **while** loop is to execute a statement or code block repeatedly as long as an **expression** is true. Once the expression becomes **false,** the loop terminates.

Syntax:

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

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var count = 0;
        document.write("Starting Loop ");
        while (count < 10){
          document.write("Current Count : " + count + "<br />");
          count++;
        }
        document.write("Loop stopped!");
    </script>
  </body>
</html>
```

b. <u>The do...while Loop:</u>

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;
}
```

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var count = 0;
        document.write("Starting Loop" + "<br />");
        do{
          document.write("Current Count : " + count + "<br />");
          count++;
        }
        while (count < 5);
        document.write ("Loop stopped!");
    </script>
  </body>
</html>
```

c. <u>For Loop:</u>

The '**for**' loop is the most compact form of looping. It 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 a given condition is true or not. If the condition is true, then the code given inside the loop will be executed, otherwise the control will come out of the loop.
- The **iteration statement** where we can increase or decrease our counter.

We can put all the three parts in a single line separated by semicolons.

Syntax:

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

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var count;
        document.write("Starting Loop" + "<br />");
        for(count = 0; count < 10; count++){
          document.write("Current Count : " + count );
          document.write("<br />");
```

```
        }
        document.write("Loop stopped!");
    </script>
  </body>
</html>
```

d. For… in Loop:

The **for...in** loop is used to loop through an object's properties. As we have not discussed Objects yet, we may not feel comfortable with this loop. But once we understand how objects behave in JavaScript, we will find this loop very useful.

Syntax

```
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.

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var aProperty;
        document.write("Navigator Object Properties<br /> ");
        for (aProperty in navigator) {
          document.write(aProperty);
          document.write("<br />");
        }
        document.write ("Exiting from the loop!");
    </script>
  </body>
</html>
```

e. Loop Control:

JavaScript provides full control to handle loops and switch statements. There may be a situation when we need to come out of a loop without reaching its bottom. There may also be a situation when we want to skip a part of our code block and start the next iteration of the loop.

To handle all such situations, JavaScript provides ***break*** and ***continue*** statements. These statements are used to immediately come out of any loop or to start the next iteration of any loop respectively.

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.

Example:

```
<html>
  <body>
    <script type="text/javascript">
      var x = 1;
      document.write("Entering the loop<br /> ");
      while (x < 20)
      {
        if (x == 5){
          break; // breaks out of loop completely
        }
        x = x + 1;
        document.write( x + "<br />");
      }
      document.write("Exiting the loop!<br /> ");
    </script>
  </body>
</html>
```

- **The continue Statement:**

The **continue** statement tells the interpreter to immediately start the next iteration of the loop and skip the remaining code block. When a **continue** statement is encountered, the program flow moves to the loop check expression immediately and if the condition remains true, then it starts the next iteration, otherwise the control comes out of the loop.

Example:

```
<html>
  <body>
    <script type="text/javascript">
        var x = 1;
        document.write("Entering the loop<br /> ");
        while (x < 10)
        {
          x = x + 1;
          if (x == 5){
            continue; // skip rest of the loop body
          }
          document.write( x + "<br />");
        }
        document.write("Exiting the loop!<br /> ");
    </script>
  </body>
</html>
```

Using Labels to Control the Flow:

Starting from JavaScript 1.2, a label can be used with **break** and **continue** to control the flow more precisely. A **label** is simply an identifier followed by a colon (:) that is applied to a statement or a block of code.

Note: Line breaks are not allowed between the **'continue'** or **'break'** statement and its label name. Also, there should not be any other statement in between a label name and associated loop.

Example 1: Break

```
<html>
  <body>
    <script type="text/javascript">
        document.write("Entering the loop!<br /> ");
        outerloop: // This is the label name
        for (var i = 0; i < 5; i++)
        {
          document.write("Outerloop: " + i + "<br />");
          innerloop:
          for (var j = 0; j < 5; j++)
          {
            if (j > 3 ) break ; // Quit the innermost loop
            if (i == 2) break innerloop; // Do the same thing
            if (i == 4) break outerloop; // Quit the outer loop
            document.write("Innerloop: " + j + " <br />");
          }
        }

        document.write("Exiting the loop!<br /> ");
    </script>
  </body>
</html>
```

Example 2: Continue

```
<html>
  <body>
    <script type="text/javascript">
      document.write("Entering the loop!<br /> ");
      outerloop: // This is the label name
      for (var i = 0; i < 3; i++)
      {
        document.write("Outerloop: " + i + "<br />");
        for (var j = 0; j < 5; j++)
        {
          if (j == 3){
            continue outerloop;
```

```
            }
            document.write("Innerloop: " + j + "<br />");
          }
        }
        document.write("Exiting the loop!<br /> ");
      </script>
    </body>
</html>
```

9. Dialog Boxes/Popup Boxes:

JavaScript supports three important types of dialog boxes. These dialog boxes can be used to raise and alert, or to get confirmation on any input or to have a kind of input from the users.

1. Alert Dialog Box:

An alert dialog box is mostly used to give a warning message to the users. For example, if one input field requires to enter some text but the user does not provide any input, then as a part of validation, we can use an alert box to give a warning message.

Nonetheless, an alert box can still be used for friendlier messages. Alert box gives only one button "OK" to select and proceed.

Example

```
<html>
  <head>
    <script type="text/javascript">
        function Warn() {
          alert ("This is a warning message!");
          document.write ("This is a warning message!");
        }
    </script>
  </head>
  <body>
    <p>Click the following button to see the result: </p>
    <form>
      <input type="button" value="Click Me" onclick="Warn();" />
    </form>
  </body>
</html>
```

2. 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**.

If the user clicks on the OK button, the window method **confirm()** will return true. If the user clicks on the Cancel button, then **confirm()** returns false.

Example

```
<html>
  <head>
    <script type="text/javascript">
        function getConfirmation(){
          var retVal = confirm("Do you want to continue ?");
          if( retVal == true ){
            document.write ("User wants to continue!");
            return true;
          }
          else{
            document.write ("User does not want to continue!");
            return false;
          }
        }
    </script>
  </head>
  <body>
    <p>Click the following button to see the result: </p>
    <form>
      <input type="button" value="Click Me" onclick="getConfirmation();" />
    </form>
  </body>
</html>
```

3. **Prompt Dialog Box:**

The prompt dialog box is very useful when we want to pop-up a text box to get user input. Thus, it enables us to interact with the user. The user needs to fill in the field and then click OK.

This dialog box is displayed using a method called **prompt()** which takes two parameters:

1. A label which we want to display in the text box
2. A default string to display in the text box.

This dialog box has two buttons: **OK** and **Cancel**. If the user clicks the OK button, the window method **prompt()** will return the entered value from the text box. If the user clicks the Cancel button, the window method **prompt()** returns **null**.

Example:

```
<html>
  <head>
    <script type="text/javascript">
        function getValue(){
          var retVal = prompt("Enter your name : ", "your name here");
          document.write("You have entered : " + retVal);
        }
    </script>
```

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

JAVASCRIPT FUNCTION:

A function is a group of reusable code which can be called anywhere in our program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Like any other advanced programming language, JavaScript also supports all the features necessary to write modular code using functions. JavaScript allows us to write our own functions as well.

1. FUNCTION DEFINITION:

Before we use a function, we need to define it. 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.

Syntax

```
<script type="text/javascript">
    function functionname(parameter-list)
    {
      statements
    }
</script>
```

Example

```
<script type="text/javascript">
    function sayHello()
    {
      alert("Hello there");
    }
</script>
```

2. CALLING A FUNCTION:

To invoke a function somewhere later in the script, we would simply need to write the name of that function as shown in the following code.

Example:

```
<html>
  <head>
    <script type="text/javascript">
      function sayHello()
      {
        document.write ("Hello there!");
      }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="sayHello()" value="Say Hello">
    </form>
    <p>Use different text in write method and then try...</p>
  </body>
</html>
```

3. Function Parameters:

Till now, we have seen functions without parameters. But there is a facility to pass different parameters while calling a function. These passed parameters can be captured inside the function and any manipulation can be done over those parameters. A function can take multiple parameters separated by comma.

Example:

```
<html>
  <head>
    <script type="text/javascript">
      function sayHello(name, age)
      {
        document.write (name + " is " + age + " years old.");
      }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="sayHello('Zara', 7)" value="Say Hello">
    </form>
    <p>Use different parameters inside the function and then try...</p>
  </body>
</html>
```

4. The return Statement:

A JavaScript function can have an optional **return** statement. This is required if we want to return a value from a function. This statement should be the last statement in a function.

For example, we can pass two numbers in a function and then we can expect the function to return their multiplication in our calling program.

Example:

```
<html>
  <head>
    <script type="text/javascript">
      function concatenate(first, last)
      {
        var full;
        full = first + last;
        return full;
      }
      function secondFunction()
      {
        var result;
        result = concatenate('Zara', 'Ali');
        document.write (result );
      }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="secondFunction()" value="Call Function">
    </form>
    <p>Use different parameters inside the function and then try...</p>
  </body>
</html>
```

5. Nested Function:

Prior to JavaScript 1.2, function definition was allowed only in top level global code, but JavaScript 1.2 allows function definitions to be nested within other functions as well. Still there is a restriction that function definitions may not appear within loops or conditionals. These restrictions on function definitions apply only to function declarations with the function statement.

As we'll discuss later in the next chapter, function literals (another feature introduced in JavaScript 1.2) may appear within any JavaScript expression, which means that they can appear within **if** and other statements.

Example:

```
<html>
  <head>
    <script type="text/javascript">
        function hypotenuse(a, b) {
          function square(x) { return x*x; }
          return Math.sqrt(square(a) + square(b));
        }
        function secondFunction(){
          var result;
          result = hypotenuse(1,2);
          document.write ( result );
        }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="secondFunction()" value="Call Function">
    </form>
    <p>Use different parameters inside the function and then try...</p>
  </body>
</html>
```

6. Function () Constructor:

The function statement is not the only way to define a new function; we can define our function dynamically using Function() constructor along with the new operator.

Note: Constructor is a terminology from Object Oriented Programming. We may not feel comfortable for the first time, which is OK.

Syntax:

```
<script type="text/javascript">
    var variablename = new Function(Arg1, Arg2..., "Function Body");
</script>
```

The **Function()** constructor expects any number of string arguments. The last argument is the body of the function – it can contain arbitrary JavaScript statements, separated from each other by semicolons.

Notice that the **Function()** constructor is not passed any argument that specifies a name for the function it creates. The **unnamed** functions created with the **Function()** constructor are called **anonymous** functions.

Example:

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

```
    <script type="text/javascript">
        var func = new Function("x", "y", "return x*y;");
        function secondFunction(){
          var result;
          result = func(10,20);
          document.write ( result );
        }
    </script>
  </head>
  <body>
    <p>Click the following button to call the function</p>
    <form>
      <input type="button" onclick="secondFunction()" value="Call Function">
    </form>
    <p>Use different parameters inside the function and then try...</p>
  </body>
</html>
```

7. Calling Function with Timer:

In JavaScript the timer is a very important feature, it allows us to execute a JavaScript function after a specified period, thereby making it possible to add a new dimension, time, to our website. With the help of the timer, we can run a command at specified intervals, run loops repeatedly at a predefined time, and synchronize multiple events in a particular time span.

There are various methods for calling function with timer such as:

1. The setTimeout() method:

Executes code at a specified interval.

Syntax:

```
setTimeout(function, delayTime)
```

In the preceding syntax the setTimeout() method contains two parameters, function and delayTime. The function parameter specifies the method that the timer calls and the delayTime parameter specifies the number of milliseconds to wait before calling the method.

Example:

```
<!DOCTYPE HTML>
<html>
<head>
<script type = "text/JavaScript">
function timedMsg(){
    var message;
    message = setTimeout("alert('This is use of setTimeout Function')",3000);
}
</script>
</head>
```

```
<body>
<form>
<p>Click the button below and output will be seen in 3 Second</p>
<input type = "button" value = "Start" onclick = "timedMsg()"/>
</form>
</body>
</html>
```

2. The clearTimeout() method:

Deactivates or cancels the timer that is set using the setTime() method.

Syntax:

```
clearTimeout(timer)
```

In the preceding syntax, timer is a variable that is created using the setTimeout() method.

Example:

```
<!DOCTYPE HTML>
<html>
<head>
<script type = "text/JavaScript">
var message;
function setMessage(){
    message = setTimeout("alert('This is use of setTimeout Function')",3000);
}
function clearMessage(){
    clearTimeout(message);
    alert("This is use of clearTimeout Function");
}
</script>
</head>
<body>
<form>
<p>Click the button below and output will be seen in 3 Second</p>
<input type = "button" value = "Start" onclick = "setMessage()"/>
<input type = "button" value = "Stop" onclick = "clearMessage()"/>
</form>
</body>
</html>
```

3. The setInterval() Method:

The setInterval() method repeats a given function at every given time-interval.

Syntax:

```
setInterval(function, milliseconds);
```

The first parameter is the function to be executed. The second parameter indicates the length of the time-interval between each execution.

Example:

```
<!DOCTYPE HTML>
    <html>
    <head>
    <script type = "text/JavaScript">
    function timedMsg(){
        var message;
        message = setInterval("alert('This is the use of setInterval Function')",3000);
    }
    </script>
    </head>
    <body>
    <form>
    <p>Click the button below and output will be seen in 3 Second</p>
    <input type = "button" value = "Start" onclick = "timedMsg()"/>
    </form>
    </body>
    </html>
```

4. The clearInterval() Method:

The clearInterval() method stops the executions of the function specified in the setInterval() method.

Syntax:

```
clearInterval(timerVariable)
```

Example:

```
<!DOCTYPE HTML>
<html>
<head>
<script type = "text/JavaScript">
var message;
function setMessage(){
    message = setInterval("alert('This is use of setInterval Function')",3000);
}
function clearMessage(){
    clearInterval(message);
    alert("This is use of clearInterval Function");
}
</script>
</head>
<body>
```

```
<form>
<p>Click the button below and output will be seen in 3 Second</p>
<input type = "button" value = "Start" onclick = "setMessage()"/>
<input type = "button" value = "Stop" onclick = "clearMessage()"/>
</form>
</body>
</html>
```

EVENT AND EVENT HANDLER:

JavaScript's interaction with HTML is handled through events that occur when the user or the browser manipulates a page.

When the page loads, it is called an event. When the user clicks a button, that click too is an event. Other examples include events like pressing any key, closing a window, resizing a window, etc.

Developers can use these events to execute JavaScript coded responses, which cause buttons to close windows, messages to be displayed to users, data to be validated, and virtually any other type of response imaginable.

Events are a part of the Document Object Model (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code.

1. onclick Event Type:

This is the most frequently used event type which occurs when a user clicks the left button of his mouse. We can put our validation, warning etc., against this event type.

Example:

```
<html>
  <head>
    <script type="text/javascript">
        function sayHello() {
          alert("Hello World")
        }
    </script>
      </head>
  <body>
    <p>Click the following button and see result</p>
    <form>
      <input type="button" onclick="sayHello()" value="Click Me" />
    </form>
  </body>
</html>
```

2. onsubmit Event type:

onsubmit is an event that occurs when we try to submit a form. We can put our form validation against this event type.

Example

The following example shows how to use onsubmit. Here we are calling a **validate()** function before submitting a form data to the webserver. If **validate()** function returns true, the form will be submitted, otherwise it will not submit the data.

```
<html>
  <head>
    <script type="text/javascript">
        function validation() {
          all validation goes here
          .........
          return either true or false
        }
    </script>
  </head>
    <form method="POST" action="t.cgi" onsubmit="return validate()">
      .......
      <input type="submit" value="Submit" />
    </form>
  </body>
</html>
```

3. onmouseover and onmouseout:

These two event types will help us to create nice effects with images or even with text as well. The onmouseover event triggers when we bring our mouse over any element and the onmouseout triggers when we move our mouse out from that element.

Example:

```
<html>
  <head>
    <script type="text/javascript">
        function over() {
          document.write ("Mouse Over");
        }

        function out() {
          document.write ("Mouse Out");
        }

    </script>
  </head>
```

```
  <body>
    <p>Bring your mouse inside the division to see the result:</p>
    <div onmouseover="over()" onmouseout="out()">
      <h2> This is inside the division </h2>
    </div>

  </body>
</html>
```

4. HTML 5 Standard Events:

The standard HTML 5 events are listed here for our reference. Here script indicates a JavaScript function to be executed against that event.

Attribute	Value	Description
Offline	script	Triggers when the document goes offline
Onabort	script	Triggers on an abort event
onafterprint	script	Triggers after the document is printed
onbeforeonload	script	Triggers before the document loads
onbeforeprint	script	Triggers before the document is printed
onblur	script	Triggers when the window loses focus
oncanplay	script	Triggers when media can start play, but might has to stop for buffering
oncanplaythrough	script	Triggers when media can be played to the end, without stopping for buffering
onchange	script	Triggers when an element changes
onclick	script	Triggers on a mouse click
oncontextmenu	script	Triggers when a context menu is triggered
ondblclick	script	Triggers on a mouse double-click
ondrag	script	Triggers when an element is dragged
ondragend	script	Triggers at the end of a drag operation
ondragenter	script	Triggers when an element has been dragged to a valid drop target
ondragleave	script	Triggers when an element is being dragged over a valid drop target
ondragover	script	Triggers at the start of a drag operation
ondragstart	script	Triggers at the start of a drag operation
ondrop	script	Triggers when dragged element is being dropped
ondurationchange	script	Triggers when the length of the media is changed
onemptied	script	Triggers when a media resource element suddenly becomes empty.
onended	script	Triggers when media has reach the end
onerror	script	Triggers when an error occur
onfocus	script	Triggers when the window gets focus
onformchange	script	Triggers when a form changes
onforminput	script	Triggers when a form gets user input

onhaschange	script	Triggers when the document has change
oninput	script	Triggers when an element gets user input
oninvalid	script	Triggers when an element is invalid
onkeydown	script	Triggers when a key is pressed
onkeypress	script	Triggers when a key is pressed and released
onkeyup	script	Triggers when a key is released
onload	script	Triggers when the document loads
onloadeddata	script	Triggers when media data is loaded
onloadedmetadata	script	Triggers when the duration and other media data of a media element is loaded
onloadstart	script	Triggers when the browser starts to load the media data
onmessage	script	Triggers when the message is triggered
onmousedown	script	Triggers when a mouse button is pressed
onmousemove	script	Triggers when the mouse pointer moves
onmouseout	script	Triggers when the mouse pointer moves out of an element
onmouseover	script	Triggers when the mouse pointer moves over an element
onmouseup	script	Triggers when a mouse button is released
onmousewheel	script	Triggers when the mouse wheel is being rotated
onoffline	script	Triggers when the document goes offline
onoine	script	Triggers when the document comes online
ononline	script	Triggers when the document comes online
onpagehide	script	Triggers when the window is hidden
onpageshow	script	Triggers when the window becomes visible
onpause	script	Triggers when media data is paused
onplay	script	Triggers when media data is going to start playing
onplaying	script	Triggers when media data has start playing
onpopstate	script	Triggers when the window's history changes
onprogress	script	Triggers when the browser is fetching the media data
onratechange	script	Triggers when the media data's playing rate has changed
onreadystatechange	script	Triggers when the ready-state changes
onredo	script	Triggers when the document performs a redo
onresize	script	Triggers when the window is resized
onscroll	script	Triggers when an element's scrollbar is being scrolled
onseeked	script	Triggers when a media element's seeking attribute is no longer true, and the seeking has ended
onseeking	script	Triggers when a media element's seeking attribute is true, and the seeking has begun
onselect	script	Triggers when an element is selected
onstalled	script	Triggers when there is an error in fetching media data
onstorage	script	Triggers when a document loads
onsubmit	script	Triggers when a form is submitted
onsuspend	script	Triggers when the browser has been fetching media data, but stopped before the entire media file was fetched
ontimeupdate	script	Triggers when media changes its playing position
onundo	script	Triggers when a document performs an undo

onunload	script	Triggers when the user leaves the document
onvolumechange	script	Triggers when media changes the volume, also when volume is set to "mute"
onwaiting	script	Triggers when media has stopped playing, but is expected to resume

5. Form Validation:

Form validation normally used to occur at the server, after the client had entered all the necessary data and then pressed the Submit button. If the data entered by a client was incorrect 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 which used to put a lot of burden on the 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 all the mandatory fields are filled in. It would require just a 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. Your code must include appropriate logic to test correctness of data.

Example:

```
<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>
```

6. Basic Form Validation:

First let us see how to do a basic form validation. In the above form, we are calling **validate ()** to validate data when **onsubmit** event is occurring. The following code shows the implementation of this validate () function.

Example:

```
<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>
```

7. Data Format Validation:

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

The following example shows how to validate an entered email address. An email address must contain at least a '@' 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.

Example:

```
<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>
```

JAVASCRIPT OBJECTS:

JavaScript is an Object Oriented Programming (OOP) language. A programming language can be called object-oriented if it provides four basic capabilities to developers:

- **Encapsulation**: the capability to store related information, whether data or methods, together in an object.
- **Aggregation**: the capability to store one object inside another object.
- **Inheritance**: the capability of a class to rely upon another class (or number of classes) for some of its properties and methods.
- **Polymorphism**: the capability to write one function or method that works in a variety of different ways.

Objects are composed of attributes. If an attribute contains a function, it is considered to be a method of the object, otherwise the attribute is considered a property.

1. Object Properties:

Object properties can be any of the three primitive data types, or any of the abstract data types, such as another object. Object properties are usually variables that are used internally in the object's methods, but can also be globally visible variables that are used throughout the page.

Syntax:

```
objectName.objectProperty = propertyValue;
```

For example: The following code gets the document title using the **"title"** property of the **document** object.

```
var str = document.title;
```

2. Object Methods:

Methods are the functions that let the object do something or let something be done to it. There is a small difference between a function and a method; a function is a standalone unit of statements and a method is attached to an object and can be referenced by the **this** keyword.

Methods are useful for everything from displaying the contents of the object to the screen to performing complex mathematical operations on a group of local properties and parameters.

For example: Following is a simple example to show how to use the **write ()** method of document object to write any content on the document.

```
document.write("This is test");
```

3. User-Defined Objects

All user-defined objects and built-in objects are descendants of an object called **Object**.

The new Operator:

The **new** operator is used to create an instance of an object. To create an object, the **new** operator is followed by the constructor method.

In the following example, the constructor methods are Object(), Array(), and Date(). These constructors are built-in JavaScript functions.

```
var employee = new Object();
var books = new Array("C++", "Perl", "Java");
var day = new Date("August 15, 1947");
```

- **The Object() Constructor:**

A constructor is a function that creates and initializes an object. JavaScript provides a special constructor function called **Object()** to build the object. The return value of the **Object()** constructor is assigned to a variable.

The variable contains a reference to the new object. The properties assigned to the object are not variables and are not defined with the **var** keyword.

Example 1: How to create an object

```
<html>
  <head>
    <title>User-defined objects</title>
    <script type="text/javascript">
      var book = new Object();   // Create the object
      book.subject = "Perl"; // Assign properties to the object
      book.author  = "Mohtashim";
    </script>
  </head>
  <body>
    <script type="text/javascript">
      document.write("Book name is : " + book.subject + "<br>");
      document.write("Book author is : " + book.author + "<br>");
    </script>
  </body>
</html>
```

Example 2: how to create an object with a User-Defined Function. Here this keyword is used to refer to the object that has been passed to a function.

```
<html>
  <head>
  <title>User-defined objects</title>
    <script type="text/javascript">
      function book(title, author){
        this.title = title;
        this.author  = author;
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
```

```
    var myBook = new book("Perl", "Mohtashim");
    document.write("Book title is : " + myBook.title + "<br>");
    document.write("Book author is : " + myBook.author + "<br>");
  </script>
 </body>
</html>
```

4. Defining Methods for an Object:

The previous examples demonstrate how the constructor creates the object and assigns properties. But we need to complete the definition of an object by assigning methods to it.

Example

The following example shows how to add a function along with an object.

```
<html>
 <head>
 <title>User-defined objects</title>

  <script type="text/javascript">
    // Define a function which will work as a method
    function addPrice(amount){
      this.price = amount;
    }

    function book(title, author){
      this.title = title;
      this.author  = author;
      this.addPrice = addPrice; // Assign that method as property.
    }
  </script>

 </head>
 <body>

  <script type="text/javascript">
    var myBook = new book("Perl", "Mohtashim");
    myBook.addPrice(100);

    document.write("Book title is : " + myBook.title + "<br>");
    document.write("Book author is : " + myBook.author + "<br>");
    document.write("Book price is : " + myBook.price + "<br>");
  </script>

 </body>
</html>
```

5. The 'with' Keyword:

The **'with'** keyword is used as a kind of shorthand for referencing an object's properties or methods.

The object specified as an argument to **with** becomes the default object for the duration of the block that follows. The properties and methods for the object can be used without naming the object.

Syntax

```
with (object){
  properties used without the object name and dot
}
```

Example:

```
<html>
  <head>
  <title>User-defined objects</title>
    <script type="text/javascript">
      // Define a function which will work as a method
      function addPrice(amount){
        with(this){
          price = amount;
        }
      }

      function book(title, author){
        this.title = title;
        this.author  = author;
        this.price = 0;
        this.addPrice = addPrice; // Assign that method as property.
      }
    </script>
  </head>
  <body>
    <script type="text/javascript">
      var myBook = new book("Perl", "Mohtashim");
      myBook.addPrice(100);

      document.write("Book title is : " + myBook.title + "<br>");
      document.write("Book author is : " + myBook.author + "<br>");
      document.write("Book price is : " + myBook.price + "<br>");
    </script>
  </body>
</html>
```

WORKING WITH BROWSER OBJECT:

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 our web page is running in.

To get information about the browser our webpage is currently running in, we use the built-in **navigator** object.

1. NAVIGATOR PROPERTIES:

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

S.No.	Property & Description
1	**appCodeName** This property is a string that contains the code name of the browser, Netscape for Netscape and Microsoft Internet Explorer for Internet Explorer.
2	**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.
3	**language** This property contains the two-letter abbreviation for the language that is used by the browser. Netscape only.
4	**mimTypes[]** This property is an array that contains all MIME types supported by the client. Netscape only.
5	**platform[]** This property is a string that contains the platform for which the browser was compiled."Win32" for 32-bit Windows operating systems
6	**plugins[]** This property is an array containing all the plug-ins that have been installed on the client. Netscape only.
7	**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.

2. NAVIGATOR METHODS:

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

S.No.	Description
1	**javaEnabled()** This method determines if JavaScript is enabled in the client. If JavaScript is enabled, this method returns true; otherwise, it returns false.
2	**plugings.refresh** This method makes newly installed plug-ins available and populates the plugins array with all new plug-in names. Netscape only.
3	**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.
4	**taintEnabled()** This method returns true if data tainting is enabled; false otherwise.

3. 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>
```

DOCUMENT OBJECT MODEL OR DOM:

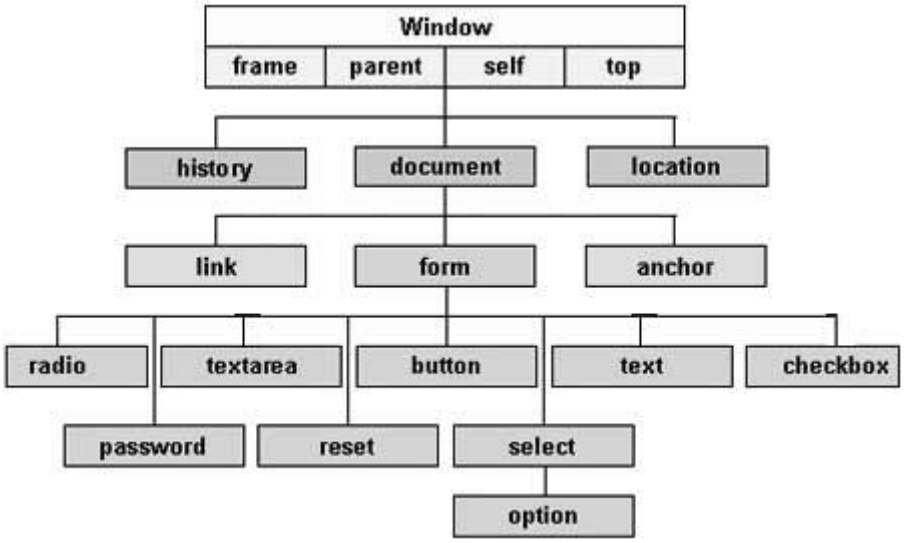
Every web page resides inside a browser window which can be considered as an object.

A Document object represents the HTML document that is displayed in that window. The Document object has various properties that refer to other objects which allow access to and modification of document content.

The way a document content is accessed and modified is called the **Document Object Model**, or **DOM**. The Objects are organized in a hierarchy. This hierarchical structure applies to the organization of objects in a Web document.

- **Window object**: Top of the hierarchy. It is the outmost element of the object hierarchy.
- **Document object**: Each HTML document that gets loaded into a window becomes a document object. The document contains the contents of the page.
- **Form object**: Everything enclosed in the <form>...</form> tags sets the form object.
- **Form control elements**: The form object contains all the elements defined for that object such as text fields, buttons, radio buttons, and checkboxes.

Here is a simple hierarchy of a few important objects:



There are several DOMs in existence. The following sections explain each of these DOMs in detail and describe how you can use them to access and modify document content.

- **The Legacy DOM**

This is the model which was introduced in early versions of JavaScript language. It is well supported by all browsers, but allows access only to certain key portions of documents, such as forms, form elements, and images.

- **The W3C DOM**

This document object model allows access and modification of all document content and is standardized by the World Wide Web Consortium (W3C). This model is supported by almost all the modern browsers.

- **The IE4 DOM**

This document object model was introduced in Version 4 of Microsoft's Internet Explorer browser. IE 5 and later versions include support for most basic W3C DOM features.

1. DOM COMPATIBILITY:

If we want to write a script with the flexibility to use either W3C DOM or IE 4 DOM depending on their availability, then we can use a capability-testing approach that first checks for the existence of a method or property to determine whether the browser has the capability you desire. For example:

```
if (document.getElementById) {
  // If the W3C method exists, use it
}
else if (document.all) {
  // If the all[] array exists, use it
}
else {
  // Otherwise use the legacy DOM
}
```

ERRORS & EXCEPTIONS HANDLING

1. TYPES OF ERRORS:

There are three types of errors in programming:

a. Syntax Errors

Syntax errors, also called **parsing errors,** occur at compile time in traditional programming languages and at interpret time in 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 the rest of the code in other threads gets executed assuming nothing in them depends on the code containing the error.

b. Runtime Errors

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

For example, the following line causes a runtime error because here the syntax is correct, but at runtime, it is trying to call a method that does not exist.

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

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

c. 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 we make a mistake in the logic that drives our script and we do not get the result as expected.

We cannot catch those errors, because it depends on our business requirement what type of logic we want to put in our program.

2. 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.

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

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 an example where we are trying to call a non-existing function which in turn is raising an exception.

```
<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>
```

Now let us try to catch this exception using **try...catch** and display a user-friendly message. We can also suppress this message, if we 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>
```

We can use **finally** block which will always execute unconditionally after the 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>
```

3. The throw Statement

We can use **throw** statement to raise our built-in exceptions or our customized exceptions. Later these exceptions can be captured and we can take an appropriate action.

Example

```
<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>
```

4. The onerror() Method

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

```
<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>
```

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.

Example

```
<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>
```

WE can display extracted information in whatever way we think it is better. We can use an **onerror** method, as shown below, to display an error message in case there is any problem in loading an image.

******

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