Chapter – 4

Introduction to JavaScript

Introduction:

- JavaScript (JS) is an object-oriented scripting language that enables us to create interactive effects on web pages.
- It is one of the three core technologies of World Wide Web Engineering:
  1. **HTML** define the content of web pages.
  2. **CSS** specify the layout of web pages.
  3. **JavaScript** programs the behavior of web pages.

Evolution:

- JavaScript, which was developed by Netscape, was originally named Mocha but soon was renamed LiveScript.
- In late 1995 LiveScript became a joint venture of Netscape and Sun Microsystems, and its name again was changed, to **JavaScript**.
- JavaScript was invented by Brendan Eich in 1995, and became an ECMA (European Computer Manufacturers Association) standard in 1997.ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

Parts of JavaScript:

- JavaScript can be divided into 3 parts:
  - **i.** **The Core JavaScript:** The core JavaScript includes its operators, expressions, statements and subprograms.
  - **ii.** **Client-side JavaScript:** It is the collection of objects that supports the control of browser and interactions with users. For example, XHTML/HTML document made responsive to user inputs such as mouse clicks, keyboard use, etc. with the help of JavaScript.
  - **iii.** **Server-side JavaScript:** It is a collection of objects that make the language useful on a Web server—for example, to support communication with a database management system.

Why JavaScript is called client-side scripting language?

- JavaScript is called client-side scripting language because it runs in web browser for interactions with users and web browsers usually run on the client. For example: JavaScript can be used to make HTML/XHTML documents response to user inputs such as mouse clicks, keyboard use, etc.
- Other examples of client-side scripting language are: Microsoft VBScript, Java Applets, Adobe Flex, etc.

Advantages of using client-side scripting language:

- Allow for **more interactivity** by immediately responding to users' actions
- **Execute quickly** because they don't require a trip to the server
- **Quick validation** of forms inputs
- May **improve the usability** of Web sites for users
- Can give developers **more control over the look and behavior** of their Web widgets
- Are **easy to manipulate** codes as well as the user interface
- Can be **used to change html elements/ attributes, css styles** and so on.

Can JavaScript be used for server-side scripting?

- **Yes**, since the advent of **Node.js** and its frameworks we can use JavaScript in both Client side and server side. Node.js is a JavaScript runtime built that uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
- MEAN (MongoDB, ExpressJS, AngularJS and Node.js) stack, used to develop web applications is also the collection of JavaScript based technologies.

Differences between Java and JavaScript:

Although there is similarity between Java and JavaScript in the syntax of their expressions, assignment statements, and control statements but there are key differences which are as follows:

- Java is an OOP programming language (mostly compiled) while Java Script is an OOP scripting language (mostly interpreted).
- Java creates applications that run in a virtual machine or browser while JavaScript code is run on a browser only.
- Java code needs to be compiled while JavaScript code need not be compiled and are all in text.
- Java is a strongly typed language. Types are all known at compile time, and operand types are checked for compatibility. Variables in JavaScript need not be declared and are dynamically typed, making compile-time type checking impossible.
- Objects in Java are static in the sense that their collection of data members and methods is fixed at compile time. JavaScript objects are dynamic: the number of data members and methods of an object can change during execution.
- They require different plug-ins.

Examples of Uses of JavaScript as client-side scripting language:

- **Interactions with users through form elements**, such as buttons and menus, can be conveniently described in JavaScript. Because button clicks and mouse movements are easily detected with JavaScript, they can be used to trigger computations and provide feedback to the user.
- **Interactions with user without forms,** which take place in dialog windows, include getting input from the user and allowing the user to make choices through buttons. It is also easy to generate new content in the browser display dynamically.

- JavaScript Document Object Model (DOM), allows JavaScript scripts to **access and modify the CSS properties and content of the elements of a displayed XHTML document**, making formally static documents highly dynamic.

Ways to embed JavaScript in XHTML/HTML document:

i. **Explicit embedding:**
JavaScript code physically residing in concerned XHTML document- may be on head or body depending on the purpose of script within desired mime type of <script> tag.
e.g
```
<script type “text/javascript”>
        ……js code…..
    </script>)
```

ii. **Implicit embedding**:
JavaScript code residing in separate **.js** file and that file included in the desired XHTML document- may be on head or body.
e.g. XHTML document:
```
<html>
  <head>
        <script type=”text/javascript” src=”jquery.js”></script>
  </head>
  <body>
        ………XHTML code…..
  </body>
</html>
```

Is JavaScript (client-side) Object Oriented Programming Language?

- Although much of its design is rooted in the concepts and approaches used in object-oriented programming language, JavaScript is not a pure object oriented programming language rather it is an **object- based language**.
- JavaScript does not have **classes**. Its objects serve both as **objects** and as models of objects.
- Without classes, JavaScript cannot have **class-based inheritance**, which is supported in object-oriented languages such as C++ and Java. It does, however, support a technique that can be used to simulate some of the aspects of inheritance. This is done with the prototype object; thus, this form of inheritance is called **prototype-based inheritance**.
- Without class-based inheritance, JavaScript cannot support **polymorphism**. A polymorphic variable can reference related methods of objects of different classes within the same class hierarchy. A method call through such a polymorphic variable can be dynamically bound to the method in the object’s class.

JavaScript Objects and Primitives:

- In JavaScript, objects are collections of properties. Each property is either a data property or a function or method property.

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the **new** keyword)
- Numbers can be objects (if defined with the **new** keyword)
- Strings can be objects (if defined with the **new** keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

All JavaScript values, except primitives, are objects.

JavaScript Primitives:

A **primitive value** is a value that has no properties or methods.

A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

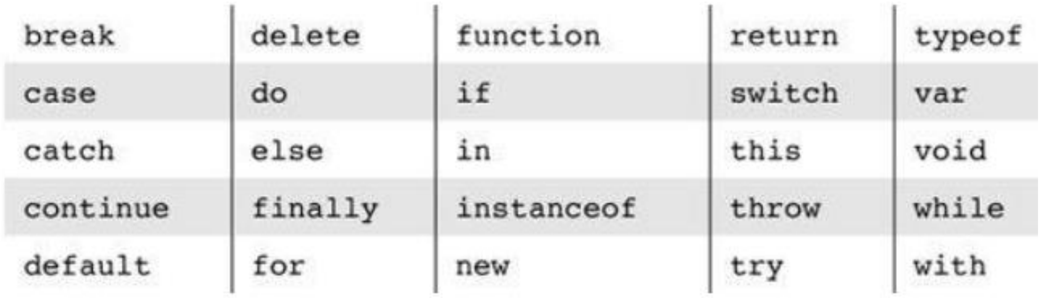
- string
- number
- boolean
- null
- undefined

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

General Syntactic Characteristics:

- JavaScript can appear directly as the content of a <script> tag.
- The type attribute of <script> must be set to “text/javascript”.
- The JavaScript script can be indirectly embedded in an XHTML document with the src attribute of a <script> tag, whose value is the name of a file that contains the script — for example,
```
  <script type = “text/javascript” src = “tst_number.js” >
 </script>
```
- The script element requires the closing tag, even though it has no content when the src attribute is included.

- In JavaScript also, identifiers, or names must begin with a letter, an underscore (_), or a dollar sign ($). Subsequent characters may be letters, underscores, dollar signs, or digits.
- There is no length limitation for identifiers. The letters in a variable name in JavaScript are case sensitive, meaning that FRIZZY, Frizzy, FrIzZy, frizzy, and friZZy are all distinct names.
- JavaScript has 25 reserved words, which are listed as below:



break	delete	function	return	typeof
case	do	if	switch	var
catch	else	in	this	void
continue	finally	instanceof	throw	while
default	for	new	try	with

- JavaScript also has a large collection of predefined words, including alert, open, java, and self.
- JavaScript has two forms of comments:
  i. First, whenever two adjacent slashes (//) appear on a line, the rest of the line is considered a comment.
  ii. Second, /* may be used to introduce a comment, and */ to terminate it, in both single- and multiple-line comments.
- The XHTML comment used to hide JavaScript uses the normal beginning syntax, <!-- to start the comment and //--> to end the comment.

Primitives, Operations and Expressions:

1. Primitive Types:
   - JavaScript has five primitive types: Number, String, Boolean, Undefined, and Null.
   - Each primitive value has one of these types.
   - JavaScript includes predefined objects that are closely related to the Number, String, and Boolean types, named **Number**, **String**, and **Boolean**, respectively.
   - These objects are called **wrapper** objects.
   - Each contains a property that stores a value of the corresponding primitive type. The purpose of the wrapper objects is to provide properties and methods that are convenient for use with values of the primitive types.
   - In the case of Number, the properties are more useful; in the case of String, the methods are more useful.
   - Because JavaScript coerces values between the Number type primitive values and Number objects and between the String type primitive values and String objects, the methods of Number and String can be used on variables of the corresponding primitive types.
   - In fact, in most cases you can simply treat Number and String type values as if they were objects.

- Objects are referenced types. The difference between primitives and objects is shown in the following example. Suppose that prim is a primitive variable with the value 17 and obj is a Number object whose property value is 17

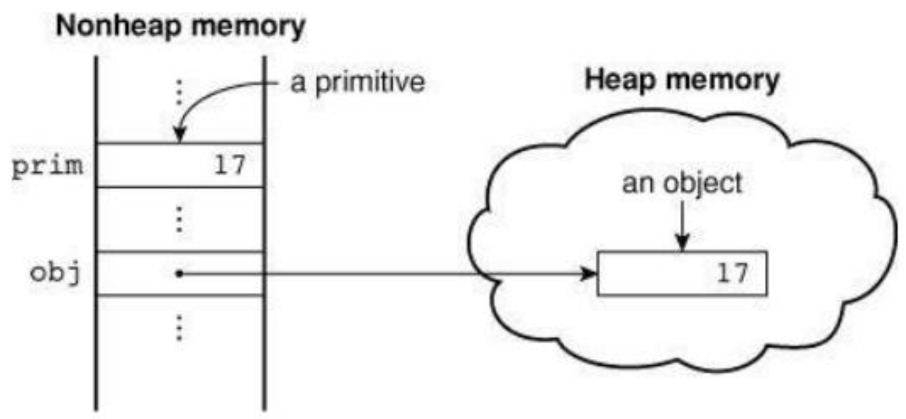


Figure 4.1 Primitives and objects

2. <u>Number and String Literals:</u>
    - All numeric literals are values of type Number.
    - The Number type values are represented internally in double-precision floating-point form.
    - Because of this single numeric data type, numeric values in JavaScript are often called numbers.
    - Literal numbers in a script can have the forms of either integer or floating-point values.
      Integer literals are strings of digits. Floating-point literals can have decimal points, exponents, or both. Exponents are specified with an uppercase or lowercase e and a possibly signed integer literal.
    - A string literal is a sequence of zero or more characters delimited by either single quotes (‘) or double quotes (“).

3. <u>Other primitive types:</u>
    - The only value of type **Null** is the reserved word null, which indicates no value.
    - A variable is **null** if it has not been explicitly declared or assigned a value.
    - If an attempt is made to use the value of a variable whose value is **null**, it will cause a runtime error.
    - The only value of type **Undefined** is undefined. It is not defined by programmer as variable value.Unlike null, there is no reserved word undefined.
    - If a variable has been explicitly declared, but not assigned a value, it has the value **undefined**.
    - If the value of an undefined variable is displayed, the word “**undefined**” is displayed.
    - The only values of type Boolean are **true** and **false**. These values are usually computed as the result of evaluating a relational or Boolean expression.

Question:

Q. Differentiate between **null** and **undefined** in JavaScript with example.

Example:

UNDEFINED:

var TestVar;

alert(TestVar); //shows undefined

alert(typeof TestVar); //shows undefined

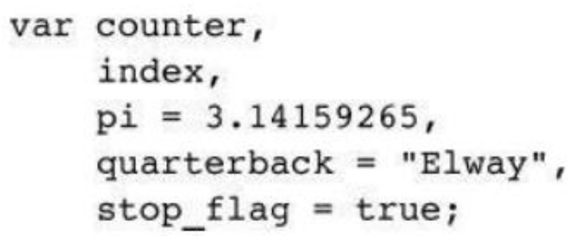
NULL:

var TestVar = null;

alert(TestVar); //shows null

alert(typeof TestVar); //shows object

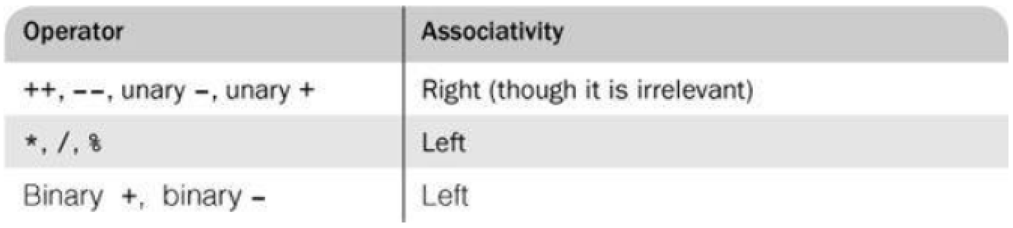
4. Declaring Variables:
   - One of the characteristics of JavaScript that sets it apart from most common non-scripting programming languages is that it is dynamically typed. This means that a variable can be used for anything.
   - Variables are not typed; values are. A variable can have the value of any primitive type, or it can be a reference to any object.
   - The type of the value of a particular appearance of a variable in a program can be determined by the interpreter.
   - In many cases, the interpreter converts the type of a value to whatever is needed for the context in which it appears.
   - A variable can be declared either by assigning it a value, in which case the interpreter implicitly declares it to be a variable, or by listing it in a declaration statement that begins with the reserved word **var**.
   - Initial values can be included in a **var** declaration, as with some of the variables in the following declaration:



```
var counter,
    index,
    pi = 3.14159265,
    quarterback = "Elway",
    stop_flag = true;
```

5. Numeric Operators:
   - JavaScript has the typical collection of numeric operators:
     the binary operators + for addition,
     - for subtraction,
     * for multiplication,
     / for division, and
     % for modulus.
   - The unary operators are plus (+), negate (-), decrement (--), and increment (++).
   - The increment and decrement operators can be either **prefix** or **postfix**.
   - As with other languages that have the increment and decrement unary operators, the prefix and postfix uses are not always equivalent.
   - Consider an expression consisting of a single variable and one of these operators. If the operator precedes the variable, the value of the variable is changed and the expression evaluates to the new value.
   - If the operator follows the variable, the expression evaluates to the current value of the variable and then the value of the variable is changed. For example, if the variable a has the value 7, the value of the following expression is 24:
     - (++a) * 3
     - But the value of the following expression is 21:
       (a++) * 3
       In both cases, a is set to 8.
   - The **associativity rules** of a language specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression.

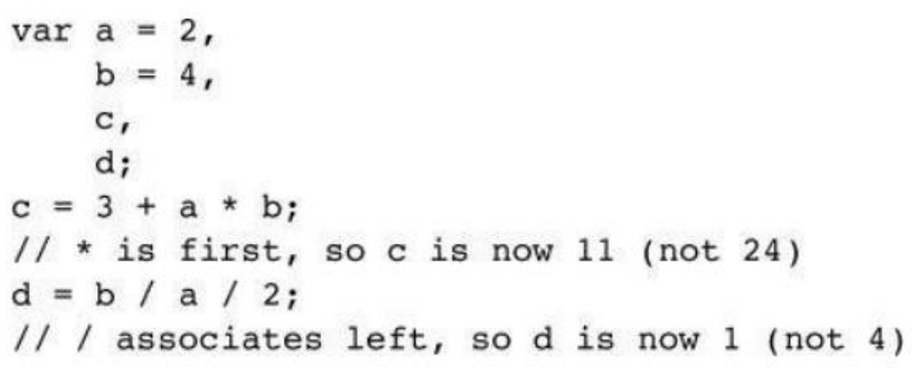
Table 4.2 Precedence and associativity of the numeric operators



Operator	Associativity
++, --, unary -, unary +	Right (though it is irrelevant)
*, /, %	Left
Binary +, binary -	Left

The first operators listed have the highest precedence.

For example:



```
var a = 2,
    b = 4,
    c,
    d;
c = 3 + a * b;
// * is first, so c is now 11 (not 24)
d = b / a / 2;
// / associates left, so d is now 1 (not 4)
```

6. <u>The **Math** Object:</u>
    - The **Math** object provides a collection of properties of **Number** objects and methods that operate on **Number** objects.
    - The **Math** object has methods for the trigonometric functions, such as **sin** (for sine) and **cos** (for cosine), as well as for other commonly used mathematical operations.
    - Among these are **floor**, to truncate a number; **round**, to round a number; and **max**, to return the largest of two given numbers.

7. <u>The **Number** Object:</u>
    - The **Number** object includes a collection of useful properties that have constant values. There is a list of properties that are referenced through **Number**. For example,
      Number.MIN_VALUE
      references the smallest representable number on the computer being used.
    - Any arithmetic operation that results in an error (e.g., division by zero) or that produces a value that cannot be represented as a double-precision floating-point number, such as a number that is too large (an overflow), returns the value "not a number," which is displayed as NaN.
    - If NaN is compared for equality against any number, the comparison fails.
    - Surprisingly, in a comparison, NaN is not equal to itself.
    - To determine whether a variable has the NaN value, the predefined predicate function isNaN() must be used. For example, if the variable a has the NaN value, isNaN(a) returns true.
    - The Number object has a method, toString, which it inherits from Object but overrides.
    - The toString method converts the number through which it is called to a string. Because numeric primitives and Number objects are always coerced to the other when necessary, toString can be called through a numeric primitive, as in the following code:
      var price = 427,
      str_price;
      ...
      str_price = price.toString();

Properties of Number:

Property	**Meaning**
MAX_VALUE	Largest representable number on the computer being used
MIN_VALUE	Smallest representable number on the computer
NaN	Not a number
POSITIVE_INFINITY	Special value to represent infinity
NEGATIVE_INFINITY	Special value to represent negative infinity
PI	The value of π

8. The String Concatenation Operator:
   JavaScript string catenation is specified with the operator denoted by a plus sign (+). For example, if the value of **first** is “Pokhara”, the value of the following expression is “Pokhara University”:
   first + “ University”

9. Implicit Type Conversion:
   - The JavaScript interpreter performs several different implicit type conversions. Such conversions are called **coercions**.
   - In general, when a value of one type is used in a position that requires a value of a different type, JavaScript attempts to convert the value to the type that is required.
   - The most common examples of these conversions involve **primitive string** and **number values.**
   - If either operand of a + operator is a string, the operator is interpreted as a string catenation operator. If the other operand is not a string, it is coerced to a string. For example, consider the following expression:
     “August ” + 1977
     In this expression, because the left operand is a string, the operator is considered to be a catenation operator. This forces string context on the right operand, so the right operand is implicitly converted to a string. Therefore, the expression evaluates to
     “August 1997”
   - Now consider the following expression:
     7 * “3”
     In this expression, the operator is one that is used only with numbers. This forces a numeric context on the right operand. Therefore, JavaScript attempts to convert it to a number. In this example, the conversion succeeds, and the value of the expression is 21.

10. Explicit Type Conversions:

- o There are several different ways to force type conversions, primarily between strings and numbers. Strings that contain numbers can be converted to numbers with the `String` constructor, as in the following code:
  var str_value = String(value);

- o This conversion could also be done with the `toString` method, which has the advantage that it can be given a parameter to specify the base of the resulting number (although the base of the number to be converted is taken to be decimal). An example of such a conversion is:
  var num = 6;
  var str_value = num.toString();
  var str_value_binary = num.toString(2);

In the first conversion, the result is `"6"`; in the second, it is `"110"`.

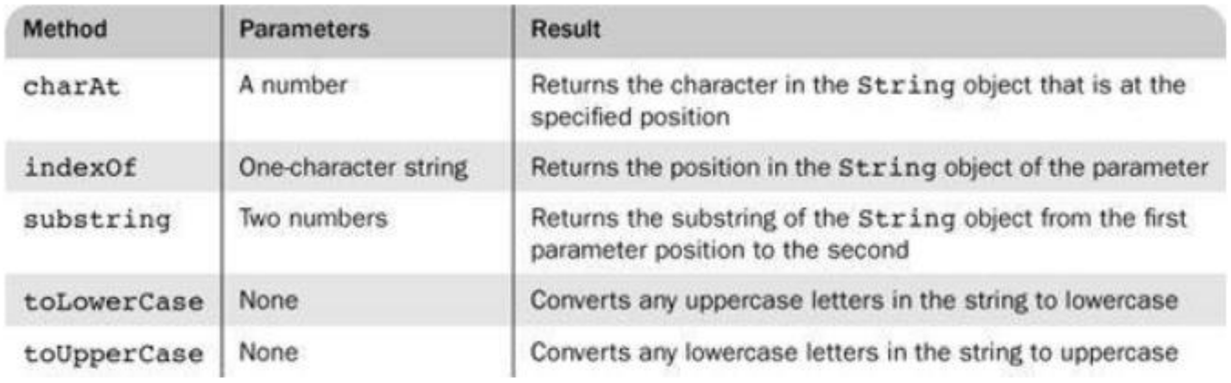
- o A number also can be converted to a string by catenating it with the empty string. Strings can be explicitly converted to numbers in several different ways. One way is with the `Number` constructor, as in the following statement:
  var number = Number(aString);
- o The same conversion could be specified by subtracting zero from the string, as in the following statement:
  var number = aString - 0;
- o Both of these conversions have the following restriction: The number in the string cannot be followed by any character except a space.
- o For example, if the number happens to be followed by a comma, the conversion will not work. JavaScript has two predefined string functions that do not have this problem.
- o The two, `parseInt` and `parseFloat`, are not `String` methods, so they are not called through `String` objects; however, they operate on the strings given as parameters.
- o The `parseInt` function searches its string parameter for an integer literal. If one is found at the beginning of the string, it is converted to a number and returned.
- o If the string does not begin with a valid integer literal, `NaN` is returned. The `parseFloat` function is similar to `parseInt`, but it searches for a floating-point literal, which could have a decimal point, an exponent, or both.
- o In both `parseInt` and `parseFloat`, the numeric literal could be followed by any nondigit character without causing any problems.

11. **String** Properties and Methods

- o Because JavaScript coerces primitive string values to and from String objects when necessary, the differences between the String object and the String type have little effect on scripts.
- o String methods can always be used through String primitive values, as if the values were objects. The String object includes one property, length, and a large collection of methods.
  The number of characters in a string is stored in the length property as follows:

var str = "George";
var len = str.length;
In this code, len is set to the number of characters in str, namely, 6. In the expression str.length, str is a primitive variable, but we treated it as if it were an object (referencing one of its properties).

- In fact, when str is used with the length property, JavaScript implicitly builds a temporary String object with a property whose value is that of the primitive variable.
- After the second statement is executed, the temporary String object is discarded.
- A few of the most commonly used `String` methods:



Method	Parameters	Result
`charAt`	A number	Returns the character in the `String` object that is at the specified position
`indexOf`	One-character string	Returns the position in the `String` object of the parameter
`substring`	Two numbers	Returns the substring of the `String` object from the first parameter position to the second
`toLowerCase`	None	Converts any uppercase letters in the string to lowercase
`toUpperCase`	None	Converts any lowercase letters in the string to uppercase

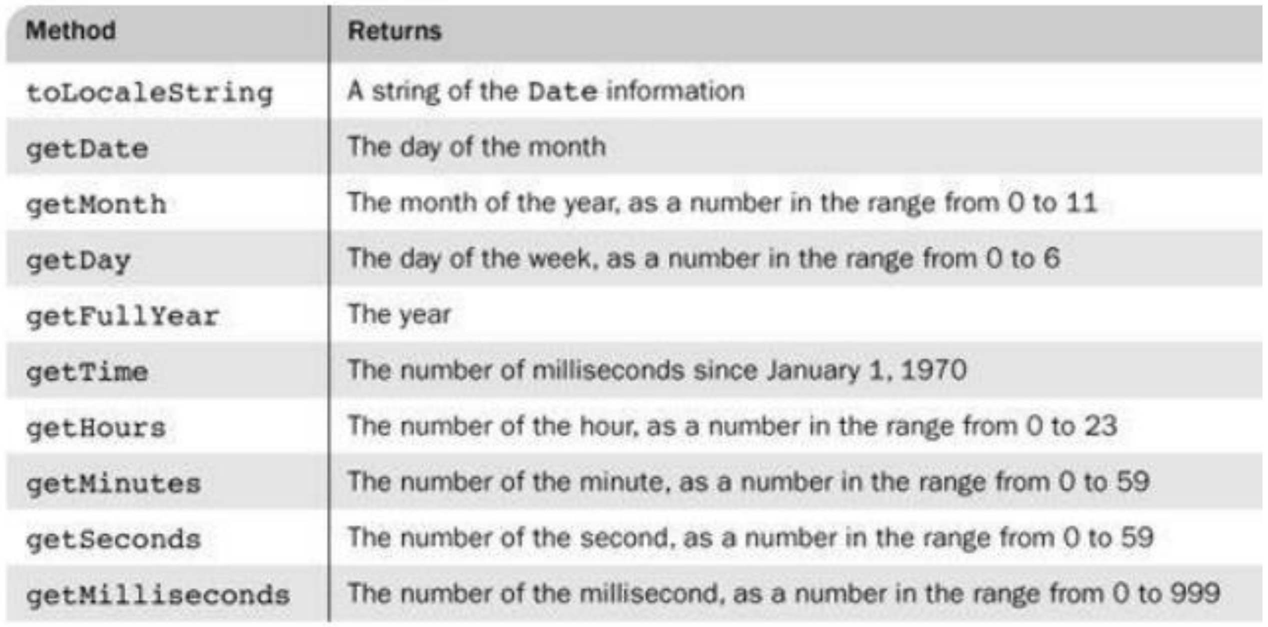
12. <u>The TypeOf Operator:</u>
    - The `typeof` operator returns the type of its single operand.
    - This operation is quite useful in some circumstances in a script. `typeof` produces "`number`","`string`", or "`boolean`" if the operand is of primitive type Number, String, or Boolean, respectively.
    - If the operand is an object or `null`, `typeof` produces "`object`".
    - If the operand is a variable that has not been assigned a value, `typeof` produces "`undefined`", reflecting the fact that variables themselves are not typed.
    - Notice that the `typeof` operator always returns a string. The operand for `typeof` can be placed in parentheses, making it appear to be a function. Therefore, `typeof x` and `typeof(x)` are equivalent.

13. <u>Assignment Statement:</u>
    Similar to C-based programming language. For example:
    a += 7;
    means the same as
    a = a + 7;

14. <u>The Date Object:</u>
    - A `Date` object is created with the `new` operator and the `Date` constructor, which has several forms. `Date` constructor, takes no parameters and builds an object with the current date and time for its properties. For example, we might have
      var today = new Date();

The date and time properties of a `Date` object are in two forms: local and Coordinated Universal Time (UTC, which was formerly named Greenwich Mean Time)

Methods of Date Object:



Method	Returns
`toLocaleString`	A string of the `Date` information
`getDate`	The day of the month
`getMonth`	The month of the year, as a number in the range from 0 to 11
`getDay`	The day of the week, as a number in the range from 0 to 6
`getFullYear`	The year
`getTime`	The number of milliseconds since January 1, 1970
`getHours`	The number of the hour, as a number in the range from 0 to 23
`getMinutes`	The number of the minute, as a number in the range from 0 to 59
`getSeconds`	The number of the second, as a number in the range from 0 to 59
`getMilliseconds`	The number of the millisecond, as a number in the range from 0 to 999

Question:

Q. Differentiate between **objects** and **primitives** in JavaScript with example.

Example:

Primitive:

```
var name=”ncit”;
console.log(name);  //output-ncit
var secondName=name;
name = “college”;
console.log(secondName); // output-ncit
```

Object:

```
var student={ name:’abc’,roll:99};
console.log(student); // output-above object as array
secondStudent=student;
console.log(secondStudent); // output- same object student
```

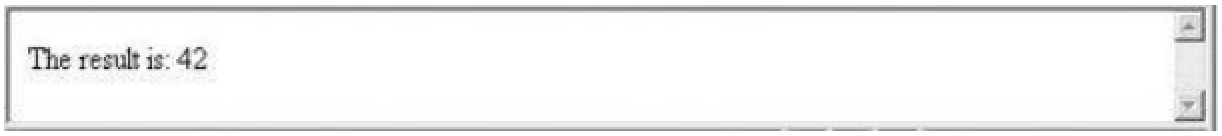
person.name='bcd';

console.log(student); // output-same above object with name changed from 'abc' to 'bcd'

Screen Output and Keyboard Input:

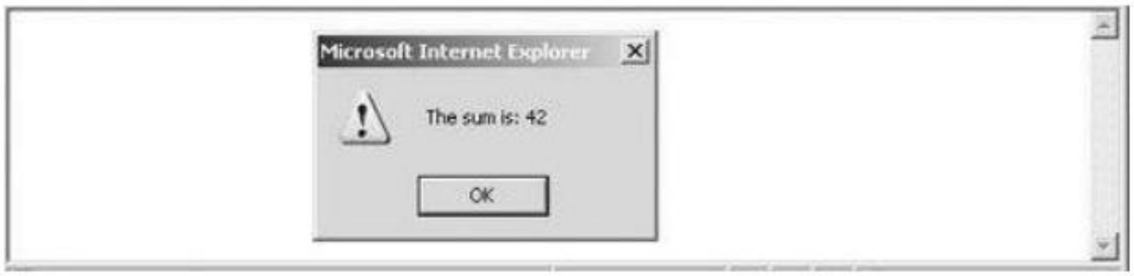
- JavaScript models the XHTML document with the `Document` object.
- The window in which the browser displays an XHTML document is modeled with the `Window` object.
- The `Window` object includes two properties, `document` and `window`. The `document` property refers to the `Document` object. The `window` property is self-referential; it refers to the `Window` object.
- The `Document` object has several properties and methods. For example: `write`
  `For example:`
  document.write("The result is: ", result, "<br />");

  outputs:



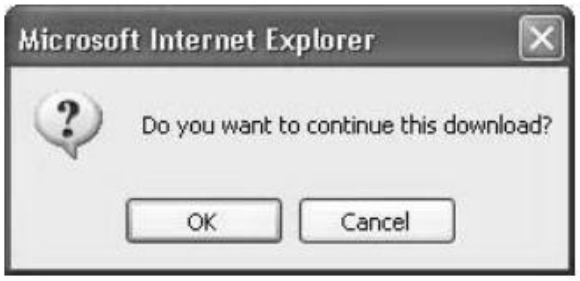
The result is: 42

- `Window` includes three methods that create dialog boxes for three specific kinds of user interactions.
- The three methods—`alert`, `confirm`, and `prompt`—which are described in the following paragraphs, are used primarily for debugging rather than as part of a servable document
- The `alert` method opens a dialog window and displays its parameter in that window. It also displays an OK button.
  For example:
  alert("The sum is:" + sum + "\n");
  output:



The sum is: 42

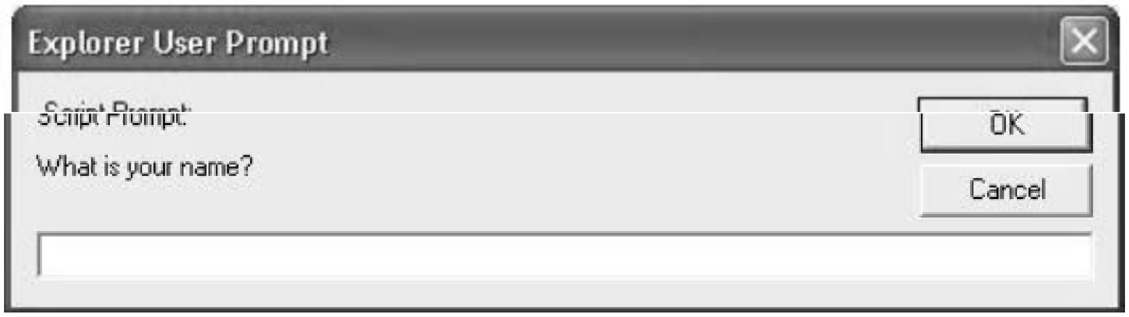
- The `confirm` method opens a dialog window in which the method displays its string parameter, along with two buttons: OK and Cancel. `confirm` returns a Boolean value that indicates the user's button input: `true` for OK and `false` for Cancel.
  var question =
  confirm("Do you want to continue this download?");



Do you want to continue this download?

- The `prompt` method creates a dialog window that contains a text box used to collect a string of input from the user, which `prompt` returns as its value. As with `confirm`, this window also includes two buttons: OK and Cancel.
  For example:
  name = prompt("What is your name?", "");

  <u>Output:</u>



Script Prompt:
What is your name?

For example (to compute the real roots of quadratic equation):

```
<?xml version = "1.0"  encoding = "utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- roots.html
     A document for roots.js
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> roots.html </title>
  </head>
  <body>
    <script type = "text/javascript"  src = "roots.js" >
    </script>
  </body>
</html>
```

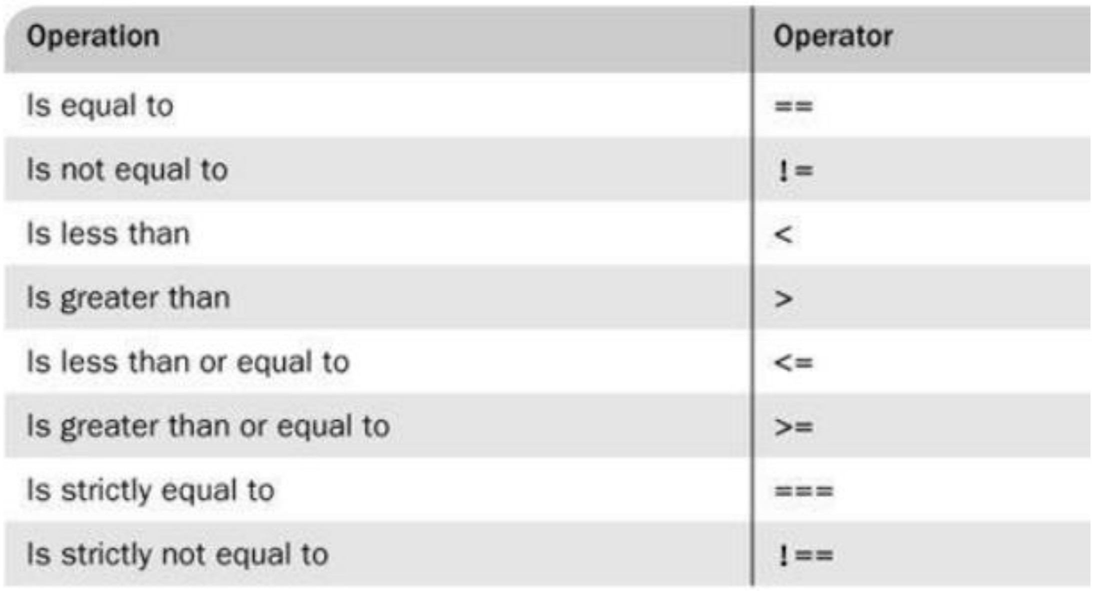
```
// roots.js
//   Compute the real roots of a given quadratic
//   equation. If the roots are imaginary, this script
//   displays NaN, because that is what results from
//   taking the square root of a negative number

// Get the coefficients of the equation from the user
var a = prompt("What is the value of 'a'? \n", "");
var b = prompt("What is the value of 'b'? \n", "");
var c = prompt("What is the value of 'c'? \n", "");

// Compute the square root and denominator of the result
var root_part = Math.sqrt(b * b - 4.0 * a * c);
var denom = 2.0 * a;

// Compute and display the two roots
var root1 = (-b + root_part) / denom;
var root2 = (-b - root_part) / denom;
document.write("The first root is: ", root1, "<br />");
document.write("The second root is: ", root2, "<br />");
```

Control Statements:

1. Control Expressions:

- The expressions upon which statement flow control can be based include primitive values, relational expressions, and compound expressions.
- The result of evaluating a control expression is one of the Boolean values `true` and `false`.
- If the value of a control expression is a string, it is interpreted as `true` unless it is either the empty string ("") or a zero string ("0").
- If the value is a number, it is `true` unless it is zero (`0`).
- If the special value, `NaN`, is interpreted as a Boolean, it is false.
- If `undefined` is used as a Boolean, it is false.
- When interpreted as a Boolean, `null` is false.
- When interpreted as a number, `true` has the value `1` and `false` has the value `0`.

Relational Operators:



Operation	Operator
Is equal to	==
Is not equal to	!=
Is less than	<
Is greater than	>
Is less than or equal to	<=
Is greater than or equal to	>=
Is strictly equal to	===
Is strictly not equal to	!==

Difference between '==' and '===':

- If the two operands are not of the same type and the operator is neither === nor !==
- Thus, the expression "3" === 3 evaluates to `false`, === checks for equality as well as data type of values. So, as "3" is a string but 3 is a number, thus it evaluates false.
- While "3" == 3 evaluates to `true` because == only checks for values, not for data types.

Comparison of Object Variables:

- Comparisons of variables that reference objects are rarely useful. If `a` and `b` reference different objects, `a == b` is never true, even if the objects have identical properties. `a == b` is true only if `a` and `b` reference the same object

Operators Precedence and Associativity

Operators	Associativity
++, --, unary -	Right
*, /, %	Left
+, -	Left
>, <, >= ,<=	Left
==, !=	Left
===,!==	Left
&&	Left
||	Left
=, +=, -=, *=, /=, &&=, ||=, %=	Right

Highest-precedence operators are listed first.

2. <u>Selection Statements:</u>

- The selection statements (`if-then` and `if-then-else`) of JavaScript are similar to those of the common programming languages.
- For example:

```
if (a > b)
    document.write("a is greater than b <br />");
else {
    a = b;
    document.write("a was not greater than b <br />",
                   "Now they are equal <br />");
}
```

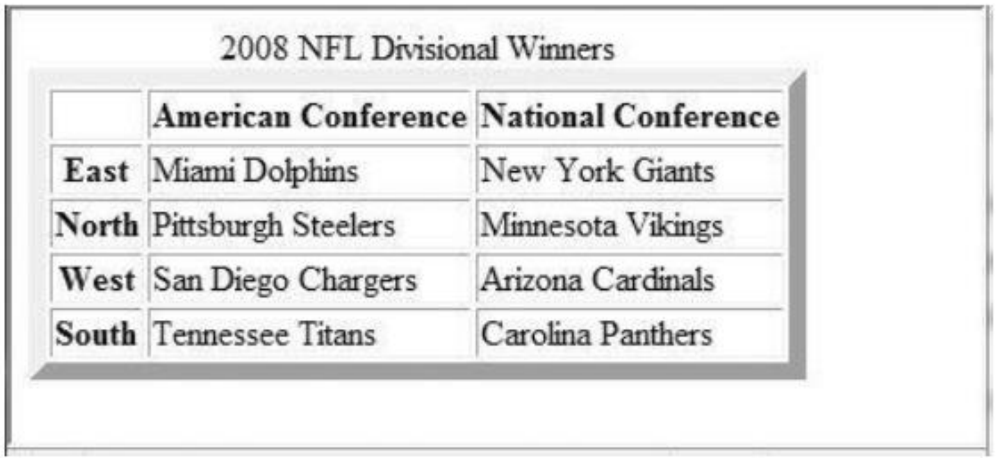
3. <u>The Switch Statement:</u>
   - JavaScript has a `switch` statement that is similar to that of the other common programming languages.

- For example:

```
switch (expression) {
     case value_1:
          // statement(s)
     case value_2:
          // statement(s)
     ...
     [default:
          // statement(s)]
}
```

- In any `case` segment, the statement(s) can be either a sequence of statements or a compound statement.
- A `break` statement appears as the last statement in each sequence of statements following a case.

Example to display table (with prompt for pixel value):



2008 NFL Divisional Winners

	American Conference	National Conference
East	Miami Dolphins	New York Giants
North	Pittsburgh Steelers	Minnesota Vikings
West	San Diego Chargers	Arizona Cardinals
South	Tennessee Titans	Carolina Panthers

JavaScript Code:

```
var bordersize;
bordersize = prompt("Select a table border size \n" +
                    "0 (no border) \n" +
                    "1 (1 pixel border) \n" +
                    "4 (4 pixel border) \n" +
                    "8 (8 pixel border) \n");

switch (bordersize) {
  case "0": document.write("<table>");
            break;
  case "1": document.write("<table border = '1'>");
            break;
  case "4": document.write("<table border = '4'>");
            break;
  case "8": document.write("<table border = '8'>");
            break;
  default:  document.write("Error - invalid choice: ",
                           bordersize, "<br />");
}

document.write("<caption> 2008 NFL Divisional",
               " Winners </caption>");
document.write("<tr>",
               "<th />",

 "<th> American Conference </th>",
 "<th> National Conference </th>",
 "</tr>",
 "<tr>",
 "<th> East </th>",
 "<td> Miami Dolphins </td>",
 "<td> New York Giants </td>",
 "</tr>",
 "<tr>",
 "<th> North </th>",
 "<td> Pittsburgh Steelers </td>",
 "<td> Minnesota Vikings </td>",
 "</tr>",
 "<tr>",
 "<th> West </th>",
 "<td> San Diego Chargers </td>",
 "<td> Arizona Cardinals </td>",
 "</tr>",
 "<tr>",
 "<th> South </th>",
 "<td> Tennessee Titans </td>",
 "<td> Carolina Panthers </td>",
 "</tr>",
 "</table>");
```

4. Loop Statements:
   - We normally use loop, when we want to run the same code over and over again, each time with a different value.

Different Kinds of Loops

JavaScript supports different kinds of loops:

- **for** - loops through a block of code a number of times
- **for/in** - loops through the properties of an object
- **while** - loops through a block of code while a specified condition is true
- **do/while** - also loops through a block of code while a specified condition is true

The For Loop:

The for loop is often the tool you will use when you want to create a loop.

The for loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {
   code block to be executed
}
```

Statement 1

- It is executed before the loop (the code block) starts, so normally used to initialize variables.
- Statement 1 is optional.
- We can initiate many values in statement 1 (separated by comma)
- We can omit statement 1 (like when your values are set before the loop starts)

For example:

```
var cars = ["BMW", "Volvo", "Saab", "Ford"];

var i, len, text;

for (i = 0, len = cars.length, text = ""; i < len; i++) {

   text += cars[i] + "<br>";

}
```

Output:

```
BMW
Volvo
Saab
Ford
```

Statement 2

- It defines the condition for running the loop (the code block).
- Often statement 2 is used to evaluate the condition of the initial variable.
- Statement 2 is also optional.
- If statement 2 returns true, the loop will start over again, if it returns false, the loop will end.
- If we omit statement 2, we must provide a **break** inside the loop. Otherwise the loop will never end.

Statement 3

- It is executed each time after the loop (the code block) has been executed.
- Often statement 3 increments the value of the initial variable.
- Statement 3 is also optional.
- Statement 3 can also do negative increment (i--), positive increment (i = i + 15), or anything else.

For example:

```
var i = 0;
var len = cars.length;
var text = "";
for (; i < len; ) {
    text += cars[i] + "<br>";
    i++;
}
```

Example to illustrate **Date** Object and **for loop** Application:

```
// Get the current date
var today = new Date();

// Fetch the various parts of the date
var dateString = today.toLocaleString();
var day = today.getDay();
var month = today.getMonth();
var year = today.getFullYear();
var timeMilliseconds = today.getTime();
var hour = today.getHours();
var minute = today.getMinutes();
var second = today.getSeconds();
var millisecond = today.getMilliseconds();

// Display the parts
document.write(
  "Date: " + dateString + "<br />",
  "Day: " + day + "<br />",
  "Month: " + month + "<br />",
  "Year: " + year + "<br />",
  "Time in milliseconds: " + timeMilliseconds + "<br />",
  "Hour: " + hour + "<br />",
  "Minute: " + minute + "<br />",
  "Second: " + second + "<br />",
  "Millisecond: " + millisecond + "<br />");

// Time a loop
var dum1 = 1.00149265, product = 1;
var start = new Date();

for (var count = 0; count < 10000; count++)
  product = product + 1.000002 * dum1 / 1.00001;

var end = new Date();
var diff = end.getTime() - start.getTime();
document.write("<br />The loop took " + diff +
               " milliseconds <br />");
```

Output:

```
Date: Tuesday, August 11, 2009 1:54:01 PM
Day: 2
Month: 7
Year: 2009
Time in milliseconds: 1250020441561
Hour: 13
Minute: 54
Second: 1
Millisecond: 561

The loop took 9 milliseconds
```

The For/In Loop

The JavaScript for/in statement loops through the properties of an object:

For example:

```
<p id="demo"></p>

<script>

        var txt = "";

        var person = {fname:"John", lname:"Doe", age:25};

        var x;

        for (x in person) {

           txt += person[x] + " ";}

        document.getElementById("demo").innerHTML = txt;

</script>
```

Output:

```
        John Doe 25
```

The While Loop

The while loop loops through a block of code as long as a specified condition is true.

Syntax

```
while (condition) {
   code block to be executed
}
```

Example

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

```
while (i < 10) {
   text += "The number is " + i;
   i++;}
```

Note: If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, and then it will repeat the loop as long as the condition is true.

Syntax

```
do {
  code block to be executed
}
while (condition);
```

Example

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested:

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

5. Object Creation and Modification:

- Objects are often created with a `new` expression, which must include a call to a constructor method.
- The constructor that is called in the `new` expression creates the properties that characterize the new object.
- In JavaScript, the `new` operator creates a blank object—that is, one with no properties. Furthermore, JavaScript objects do not have types. The constructor both creates and initializes the properties.
  The following statement creates an object that has no properties:
  var my_object = new Object();
- In this case, the constructor called is that of `Object`, which endows the new object with no properties, although it does have access to some inherited methods. The variable `my_object` references the new object.
- Calls to constructors must include parentheses, even if there are no parameters.

- ➢ The properties of an object can be accessed with dot notation, in which the first word is the object name and the second is the property name.
- ➢ The number of properties in a JavaScript object is dynamic. At any time during interpretation, properties can be added to or deleted from an object.
  For example:

```
// Create an Object object
var my_car = new Object();
// Create and initialize the make property
my_car.make = "Ford";
// Create and initialize model
my_car.model = "Contour SVT";
```

- ➢ This code creates a new object, `my_car`, with two properties: `make` and `model`.

Above object can also be created as below:

var my_car = {make: “Ford”, model: “Contour SVT”};

- ➢ Properties can be accessed in two ways. First, any property can be accessed in the same way it is assigned a value, namely, with the object-dot-property notation. Second, the property names of an object can be accessed as if they were elements of an array.

  For example:

  var prop1 = my_car.make;
  var prop2 = my_car[“make”];
- ➢ If an attempt is made to access a property of an object that does not exist, the value `undefined` is used.
- ➢ A property can be deleted with `delete`, as in the following example:
  delete my_car.model;

6. Arrays:

Array Object Creation

- `Array` objects, unlike most other JavaScript objects, can be created in two distinct ways:
- The usual way to create any object is with the `new` operator and a call to a constructor. In the case of arrays, the constructor is named `Array`:

```
var my_list = new Array(1, 2, "three", "four");
var your_list = new Array(100);
```

- In the first declaration, an `Array` object of length 4 is created and initialized. Notice that the elements of an array need not have the same type.
- In the second declaration, a new `Array` object of length 100 is created, without actually creating any elements.
- The second way to create an `Array` object is with a literal array value, which is a list of values enclosed in brackets:
  var my_list_2 = [1, 2, “three”, “four”];

Array Methods

- `Array` objects have a collection of useful methods
- The `join` method converts all of the elements of an array to strings and catenates them into a single string
- If no parameter is provided to `join`, the values in the new string are separated by commas
  For example:
  ```
  var names = new Array["Mary", "Murray", "Murphy", "Max"];
  ...
  var name_string = names.join(" : ");
  ```
  The value of `name_string` is now `“Mary : Murray : Murphy : Max”`.
- The `reverse` method reverses the order of the elements of the `Array` object through which it is called.
- The `sort` method coerces the elements of the array to become strings if they are not already strings and sorts them alphabetically.
  For example:
  names.sort();
  The value of `names` is now `[“Mary”, “Max”, “Murphy”, “Murray”]`
- The `concat` method catenates its actual parameters to the end of the `Array` object on which it is called.
  For example:

  ```
  var names = new Array["Mary", "Murray", "Murphy", "Max"];
  ...
  var new_names = names.concat("Moo", "Meow");
  ```
  `The new_names` array now has length 6, with the elements of `names`, along with `“Moo”` and `“Meow”`, as its fifth and sixth elements.

- The slice() method returns the selected elements in an array, as a new array object.The slice() method selects the elements starting at the given *start* argument, and ends at, *but does not include*, the given *end* argument.

  For example:

  var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
  var citrus = fruits.slice(1, 3);

<u>Output:</u>
Orange, Lemon
(Includes second array [1] and third array [2] element, excluding array [3])

- The `push`, `pop`, `unshift`, and `shift` methods of `Array` allow the easy implementation of stacks and queues in arrays. The `pop` and `push` methods respectively remove and add an element to the high end of an array, as in the following code:

```
var list = ["Dasher", "Dancer", "Donner", "Blitzen"];
var deer = list.pop();     // deer is "Blitzen"
list.push("Blitzen");
  // This puts "Blitzen" back on list
```

- The `shift` and `unshift` methods respectively remove and add an element to the beginning of an array.
- For example, assume that `list` is created as before, and consider the following code:
  var deer = list.shift(); // deer is now “Dasher”
  list.unshift(“Dasher”); // This puts “Dasher” back on list
- A two-dimensional array is implemented in JavaScript as an array of arrays.
  <u>For example:</u>

```
// Create an array object with three arrays as its elements
var nested_array = [[2, 4, 6], [1, 3, 5], [10, 20, 30]
                   ];
```

7. <u>Functions:</u>

- A JavaScript function is a block of code designed to perform a particular task.
- A JavaScript function is executed when "something" invokes it (calls it).

JavaScript Function Syntax

- A JavaScript function is defined with the **function** keyword, followed by a **name**, followed by parentheses **()**.
- Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).
- The parentheses may include parameter names separated by commas: **(*parameter1, parameter2, ...*)**
- The code to be executed, by the function, is placed inside curly brackets: **{}**

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

- Function **parameters** are listed inside the parentheses () in the function definition.
- Function **arguments** are the **values** received by the function when it is invoked.
- Inside the function, the arguments (the parameters) behave as local variables.
- The code inside the function will execute when "something" **invokes** (calls) the function:
    - When an event occurs (when a user clicks a button)
    - When it is invoked (called) from JavaScript code
    - Automatically (self invoked)

Function Return

When JavaScript reaches a **return statement**, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

Example

Calculate the product of two numbers, and return the result:

```
var x = myFunction(4, 3);    // Function is called, return value will end up in x

function myFunction(a, b) {
  return a * b;             // Function returns the product of a and b
}
```

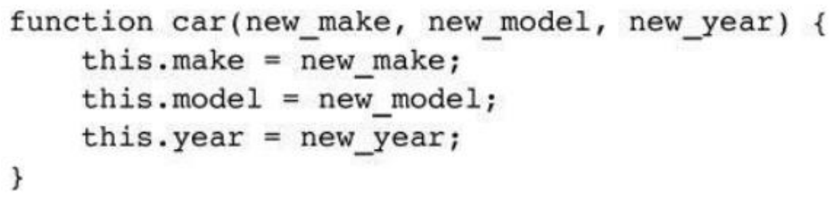
The result in x will be:

```
12
```

8. <u>Constructors:</u>

- JavaScript constructors are special methods that create and initialize the properties of newly created objects.
- Every `new` expression must include a call to a constructor whose name is the same as that of the object being created.
- Thus, Constructors are a "**blueprint**" for creating many objects of the same "type".
- Objects of the same type are created by calling the constructor function with the **new** keyword.
- Obviously, a constructor must be able to reference the object on which it is to operate.

- JavaScript has a predefined reference variable for this purpose, named `this`.
- When the constructor is called, `this` is a reference to the newly created object. The `this` variable is used to construct and initialize the properties of the object.
  For example, the constructor:



```
function car(new_make, new_model, new_year) {
    this.make = new_make;
    this.model = new_model;
    this.year = new_year;
}
```

Could be used as in the following statement:
my_car = new car (“Ford”, “Contour SVT”, “2000”);

<u>Another example:</u>

```
<p id="demo"></p>

<script>

// Constructor function for Person objects

function Person(first, last, age, eye) {

   this.firstName = first;

   this.lastName = last;

   this.age = age;

   this.eyeColor = eye;

}

// Create two Person objects

var myFather = new Person("John", "Doe", 50, "blue");

var myMother = new Person("Sally", "Rally", 48, "green");

// Display age

document.getElementById("demo").innerHTML =

"My father is " + myFather.age + ". My mother is " + myMother.age + ".";

</script>
```

9. Pattern Matching Using Regular Expressions:

- JavaScript has powerful pattern-matching capabilities based on regular expressions.
- There are two approaches to pattern matching in JavaScript: one that is based on the `RegExp` object and one that is based on methods of the `String` object.
- Patterns, which are sent as parameters to the patternmatching methods, are delimited with slashes.
- The simplest pattern-matching method is `search`, which takes a pattern as a parameter. The `search` method returns the position in the `String` object (through which it is called) at which the pattern matched. If there is no match, `search` returns `–1`.

```
var str = "Rabbits are furry";
var position = str.search(/bits/);
if (position >= 0)
    document.write("'bits' appears in position", position,
                   "<br />");
else
    document.write("'bits' does not appear in str <br />");
```

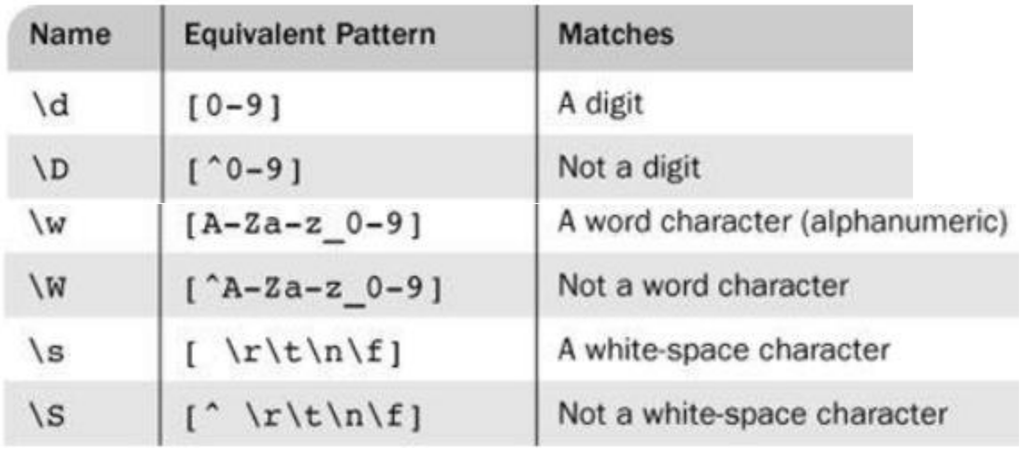
produce the following output:

```
'bits' appears in position 3
```

Character and Character-Class Patterns:

- The “normal” characters are those that are not metacharacters.
- Metacharacters are characters that have special meanings in some contexts in patterns. The following are
  the pattern metacharacters:
  \ | ( ) [ ] { } ^ $ * + ? .
- Metacharacters can themselves be matched by being immediately preceded by a backslash.
- For example, the following character class matches `'a'`, `'b'`, or `'c'`: [abc]
- The following character class matches any lowercase letter from `'a'` to `'h'`: [a-h]
- If a circumflex character (^) is the first character in a class, it inverts the specified set. For example, the following character class matches any character except the letters `'a'`, `'e'`, `'i'`, `'o'`, and `'u'`: [^aeiou]

Predefined Character- classes:



Name	Equivalent Pattern	Matches
\d	[0-9]	A digit
\D	[^0-9]	Not a digit
\w	[A-Za-z_0-9]	A word character (alphanumeric)
\W	[^A-Za-z_0-9]	Not a word character
\s	[\r\t\n\f]	A white-space character
\S	[^ \r\t\n\f]	Not a white-space character

Example of uses:

```
/\d\.\d\d/     // Matches a digit, followed by a period,
               // followed by two digits
/\D\d\D/       // Matches a single digit
/\w\w\w/       // Matches three adjacent word characters
```

- In many cases, it is convenient to be able to repeat a part of a pattern, often a character or character class. To repeat a pattern, a numeric quantifier, delimited by braces, is attached. For example, the following pattern matches `xyyyyz`: /xy{4}z/
- There are also three symbolic quantifiers: asterisk (`*`), plus (`+`), and question mark (`?`). An asterisk means zero or more repetitions, a plus sign means one or more repetitions, and a question mark means one or none. For example, the following pattern matches strings that begin with any number of `x`'s (including zero), followed by one or more `y`'s, possibly followed by `z`: /x*y+z?/

```
var str =
  "Having 4 apples is better than having 3 oranges"
var matches = str.match(/\d/g);
```

In this example, `matches` is set to `[4, 3]`.

Now consider a pattern that has parenthesized subexpressions:

```
var str = "I have 428 dollars, but I need 500";
var matches = str.match(/(\d+)([^\d]+)(\d+)/);
document.write(matches, "<br />");
```

The following is the value of the `matches` array after this code is interpreted:

```
["428 dollars, but I need 500", "428",
"dollars, but I need ", "500"]
```

- In this result array, the first element, "`428 dollars, but I need 500`", is the match; the second, third, and fourth elements are the parts of the string that matched the parenthesized parts of the pattern, `(\d+)`, `([^\d]+)`, and `(\d+)`.
- The `replace` method is used to replace substrings of the `String` object that match the given pattern. The `replace` method takes two parameters: the pattern and the replacement string.

  For example:
  var str = "Fred, Freddie, and Frederica were siblings";
  str.replace(/Fre/g, "Boy");
  In this example, `str` is set to "`Boyd, Boyddie, and Boyderica were siblings`", and `$1`, `$2`, and `$3` are all set to "`Fre`".

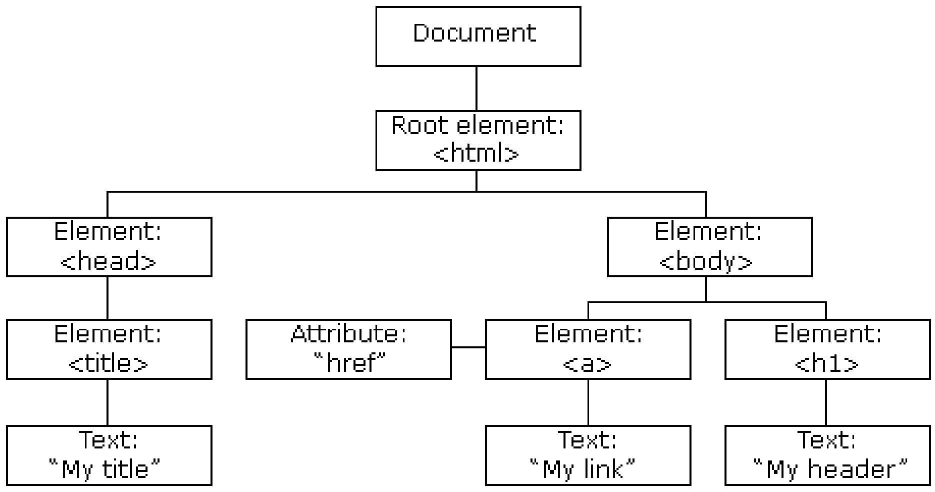
Document Object Model (DOM):

- DOM is a W3C (World Wide Web Consortium) standard that defines a standard for accessing documents.
- *"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document"*
- The W3C DOM standard is separated into 3 different parts:
  - Core DOM - standard model for all document types
  - XML DOM - standard model for XML documents
  - HTML DOM - standard model for HTML documents
- Thus, The DOM is an application programming interface (API) that defines an interface between XHTML documents and application programs. It is an abstract model because it must apply to a variety of application programming languages.

The HTML DOM (Document Object Model)

- The HTML DOM is a standard for how to get, change, add, or delete HTML elements.
- It is a standard **object** model and **programming interface** for HTML that defines:
  - The HTML elements as **objects**
  - The **properties** of all HTML elements
  - The **methods** to access all HTML elements
  - The **events** for all HTML elements
- When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.
- The **HTML DOM** model is constructed as a tree of **Objects**:

The HTML DOM Tree of Objects



Thus, with the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript **can change all the HTML elements** in the page
- JavaScript **can change all the HTML attributes** in the page
- JavaScript **can change all the CSS styles** in the page
- JavaScript **can remove existing HTML elements and attributes**
- JavaScript **can add new HTML elements and attributes**
- JavaScript **can react to all existing HTML events** in the page
- JavaScript can **create new HTML events** in the page

Methods and Properties in DOM:

- ➢ In the DOM, all HTML elements are defined as **objects**.
- ➢ The programming interface is the properties and methods of each object.
- ➢ A **property** is a value that you can get or set (like changing the content of an HTML element).
- ➢ A **method** is an action you can do (like add or deleting an HTML element).
- ➢ Thus, the data are called properties, and the operations are, naturally, called methods.
- ➢ For example, the following XHTML element would be represented as an object with two properties, type and name, with the values “text” and “address”, respectively:
  <input type = “text” name = “address”>
  In most cases, the property names in JavaScript are the same as their corresponding attribute names in XHTML.

  Example

  The following example changes the content (the innerHTML) of the <p> element with id="demo":

```
<!DOCTYPE html>

<html>

        <body>

                <h2>My First Page</h2>

                <p id="demo"></p>

                <script>

        document.getElementById("demo").innerHTML = "Hello World!";

                </script>

        </body>

</html>
```

- ➢ In the example above, getElementById is a **method**, while innerHTML is a **property**.

Element Access in JavaScript:

- The elements of an XHTML document have corresponding objects that are visible to an embedded JavaScript script. The addresses of these objects are required, both by the event handling and by the code for making dynamic changes to documents.
- There are several ways the object associated with an XHTML form element can be addressed in JavaScript.
- **The original (DOM 0) way** is to use the forms and elements arrays of the Document object, which is referenced through the document property of the Window object. As an example, consider the following XHTML document:

```
<html>
        <head>Access to form elements</head>
  <body>
        <form name=”myForm” action = “”>
                <input type=”button” name=”turnItOn” id=”turnItOn”/>
        </form>
  </body>
</html>
```

  The DOM address of the button in this example, using the `forms` and `elements` arrays, is as follows:
  var dom = document.forms[0].elements[0];

  The problem with this approach to element addressing is that the DOM address is defined by address elements that could change—namely, the `forms` and `elements` arrays. For example, if a new button were added before the `turnItOn` button in the form, the DOM address shown would be wrong.

- **Another approach** to DOM addressing is to use element names. For this, the element and its enclosing elements, up to but not including the body element, must include `name` attributes. For example, in above example:

  Using the `name` attributes, the button’s DOM address is as follows:
  var dom = document.myForm.turnItOn;

- **Yet another approach** to element addressing is to use the JavaScript method `getElementById`, which is defined in DOM 1. Because an element’s identifier (`id`) is unique in the document, this approach works, regardless of how deeply the element is nested in other elements in the document.
- For example, if the id attribute of our button is set to `"turnItOn"`, the following could be used to get the DOM address of that button element:

  var dom = document.getElementById(“turnItOn”);

  The parameter of `getElementById` can be any expression that evaluates to a string.

- Because ids are most useful for DOM addressing and `name`s are required for form-processing code, form elements often have both ids and `name`s, set to the same value.
- Buttons in a group of checkboxes and radio button often share the same name. In these cases, the names of the individual buttons obviously cannot be used in their DOM addresses. Of course, each radio button and checkbox can have an id, which would make them easy to address by using `getElementById`.
- However, this approach does not provide a convenient way to search a group of radio buttons or checkboxes to determine which is checked.
- An alternative to both `name`s and ids is provided by the implicit arrays associated with each checkbox and radio button group.
- Every such group has an array, which has the same name as the group name that stores the DOM addresses of the individual buttons in the group.
- These arrays are properties of the form in which the buttons appear. To access the arrays, the DOM address of the form object must first be obtained, as in the following example:

```
<form id = "vehicleGroup">
  <input type = "checkbox"  name = "vehicles"
         value = "car" />  Car
  <input type = "checkbox"  name = "vehicles"
         value = "truck" />  Truck
  <input type = "checkbox"  name = "vehicles"
         value = "bike" />  Bike
</form>
```

- The implicit array, `vehicles`, has three elements, which reference the three objects associated with the three checkbox elements in the group.
- This array provides a convenient way to search the list of checkboxes in a group.
- The `checked` property of a checkbox object is set to `true` if the button is checked.
- For the preceding sample checkbox group, the following code would count the number of checkboxes that were checked:

```
var numChecked = 0;
var dom = document.getElementById("vehicleGroup");
for (index = 0; index < dom.vehicles.length; index++)
  if (dom.vehicles[index].checked)
    numChecked++;
```

- Radio buttons can be addressed and handled exactly as are the checkboxes.

Finding HTML Elements

Method	**Description**
document.getElementById(*id*)	Find an element by element id
document.getElementsByTagName(*name*)	Find elements by tag name
document.getElementsByClassName(*name*)	Find elements by class name
`document.querySelectorAll("p.intro");`	Find HTML Elements by CSS Selectors

`var x = document.forms["frm1"];`
`var text = "";`
`var i;`
`for (i = 0; i < x.length; i++) {`
`    text += x.elements[i].value + "<br>";`
`}`<br>`document.getElementById("demo").innerHTML = text;`
	Find HTML Element by HTML Object Collections

Changing HTML Elements

Method	**Description**
element.innerHTML = *new html content*	Change the inner HTML of an element

element.*attribute = new value*	Change the attribute value of an HTML element
element.setAttribute*(attribute, value)*	Change the attribute value of an HTML element
element.style.*property = new style*	Change the style of an HTML element

Adding and Deleting Elements

Method	**Description**
document.createElement(*element*)	Create an HTML element
document.removeChild(*element*)	Remove an HTML element
document.appendChild(*element*)	Add an HTML element
document.replaceChild(*element*)	Replace an HTML element
document.write(*text*)	Write into the HTML output stream

Events and Events Handling:

Basic Concepts:

Event-driven Programming:

- One important use of JavaScript for Web programming is to detect certain activities of the browser and the browser user and provide computation when those activities occur.
- These computations are specified with a special form of programming called event-driven programming.
- In conventional (non-event-driven) programming, the code itself specifies the order in which it is executed, although the order is usually affected by the program's input data.
- In event-driven programming, parts of the program are executed at completely unpredictable times, often triggered by user interactions with the program that is executing.

Event

- An event is a notification that something specific has occurred, either with the browser, such as the completion of the loading of a document, or because of a browser user action, such as a mouse click on a form button.
- Strictly speaking, an event is an object that is implicitly created by the browser and the JavaScript system in response to something happening.

Examples of HTML events:

- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user strokes a key

Event Handler:

- An event handler is a script that is implicitly executed in response to the appearance of an event.
- Event handlers enable a Web document to be responsive to browser and user activities.
- One of the most common uses of event handlers is to check for simple errors and omissions in user input to the elements of a form, either when they are changed or when the form is submitted. This kind of checking saves the time of sending incorrect form data to the server.
- Events are similar to exceptions in error handling.

- Both events and exceptions occur at unpredictable times, and both often require some special program actions.

Registration:

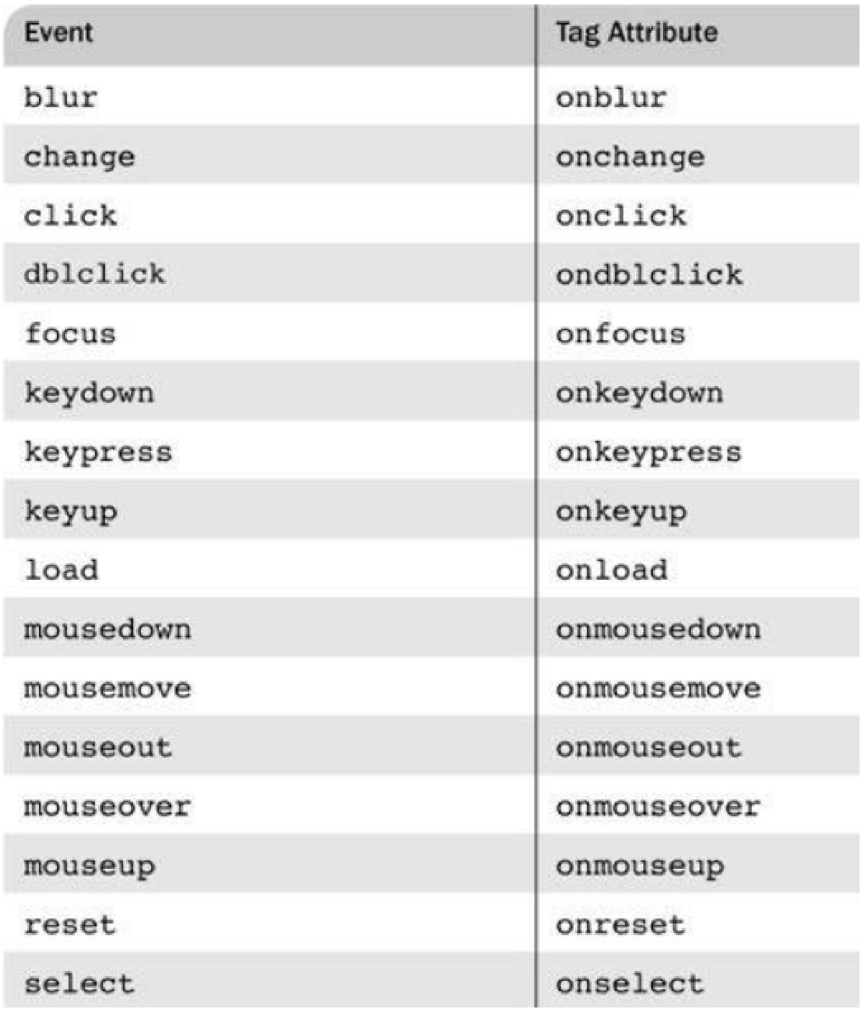
- The process of connecting an event handler to an event is called registration.
- There are two distinct approaches to event handler registration, one that assigns tag attributes and one that assigns handler addresses to object properties

Why **write** method should never be used in an event handler?

- It is because a document is displayed as its XHTML code is parsed by the browser.
- **Events usually occur after the whole document is displayed**.
- If `write` appears in an event handler, the content produced by it might be placed over the top of the currently displayed document.

Events, Attributes and Tags:

Events and their tag attributes:



Event	Tag Attribute
blur	onblur
change	onchange
click	onclick
dblclick	ondblclick
focus	onfocus
keydown	onkeydown
keypress	onkeypress
keyup	onkeyup
load	onload
mousedown	onmousedown
mousemove	onmousemove
mouseout	onmouseout
mouseover	onmouseover
mouseup	onmouseup
reset	onreset
select	onselect

submit	onsubmit
unload	onunload

Event Attributes and their tags:

Attribute	Tag	Description
onblur	`<a>`	The link loses the input focus
	`<button>`	The button loses the input focus
	`<input>`	The input element loses the input focus
	`<textarea>`	The text area loses the input focus
	`<select>`	The selection element loses the input focus
onchange	`<input>`	The input element is changed and loses the input focus
	`<textarea>`	The text area is changed and loses the input focus
	`<select>`	The selection element is changed and loses the input focus
onclick	`<a>`	The user clicks on the link
	`<input>`	The input element is clicked
ondblclick	Most elements	The user double-clicks the left mouse button
onfocus	`<a>`	The link acquires the input focus
	`<input>`	The input element receives the input focus
	`<textarea>`	A text area receives the input focus
	`<select>`	A selection element receives the input focus
onkeydown	`<body>`, form elements	A key is pressed down
onkeypress	`<body>`, form elements	A key is pressed down and released
onkeyup	`<body>`, form elements	A key is released
onload	`<body>`	The document is finished loading
onmousedown	Most elements	The user clicks the left mouse button
onmousemove	Most elements	The user moves the mouse cursor within the element
onmouseout	Most elements	The mouse cursor is moved away from being over the element
onmouseover	Most elements	The mouse cursor is moved over the element
onmouseup	Most elements	The left mouse button is unclicked
onreset	`<form>`	The reset button is clicked

onselect	`<input>`	Any text in the content of the element is selected
	`<textarea>`	Any text in the content of the element is selected
onsubmit	`<form>`	The *Submit* button is pressed
onunload	`<body>`	The user exits the document

Methods:

For single element:

```
<input type = "button" id = "myButton"
       onclick = "alert('You clicked my button!');" />
```

For more than one element at once:

```
<input type = "button" id = "myButton"
        onclick = "myButtonHandler();" />
```

Also by:

```
document.getElementById("myButton").onclick =
                                      myButtonHandler;
```

Handling Events from Body Elements

- The events most often created by body elements are `load` and `unload`.
- For example:

```
<!DOCTYPE html>
<html>
        <body onload="myFunction()">

                <h1>Hello World!</h1>

        <script>
        function myFunction() {
           alert("Page is loaded");
        }
        </script>

        </body>
</html>
```

  - The above code loads alert window with alert message when the page is loaded or refreshed.
  - Normally **onload** event can be used to check the visitor's browser type and version and load the proper version of webpage based on version, and also can be used to deal with cookies.
- Similarly, the `unload` event is probably more useful than the `load` event. It is used to do some cleanup before a document is unloaded, as when the browser user goes on to some new document. For example, if the document opened a second browser window, that window could be closed by an `unload` event handler.

Handling Events from Button Elements:

- Buttons in a Web document provide an effective way to collect simple input from the browser user.
  For example:

```
<!DOCTYPE html>
<!-- radio_click2.hmtl
     A document for radio_click2.js
     -->
<html lang = "en">
  <head>
    <title> radio_click2.html </title>
    <meta charset = "utf-8" />

    <!-- Script for the event handler -->
    <script type = "text/javascript"  src = "radio_click2.js" >
    </script>
  </head>
  <body>
    <h4> Cessna single-engine airplane descriptions </h4>
    <form id = "myForm"  action = "">
      <p>
        <label> <input type = "radio"  name = "planeButton"
                       value = "152" />
        Model 152 </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "172" />
        Model 172 (Skyhawk) </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "182" />
        Model 182 (Skylane) </label>
        <br />
        <label> <input type = "radio"  name = "planeButton"
                       value = "210" />
        Model 210 (Centurian) </label>
      </p>
    </form>

<!-- Script for registering the event handlers -->
    <script type = "text/javascript"  src = "radio_click2r.js" >
    </script>
  </body>
</html>

// radio_click2.js
//   An example of the use of the click event with radio buttons,
//   registering the event handler by assigning an event property

// The event handler for a radio button collection

function planeChoice (plane) {

// Put the DOM address of the elements array in a local variable

  var dom = document.getElementById("myForm");

// Determine which button was pressed

  for (var index = 0; index < dom.planeButton.length;
       index++) {
    if (dom.planeButton[index].checked) {
      plane = dom.planeButton[index].value;
      break;
  }
}

// Produce an alert message about the chosen airplane

  switch (plane) {
    case "152":
      alert("A small two-place airplane for flight training");
      break;
    case "172":
      alert("The smaller of two four-place airplanes");
      break;
    case "182":
      alert("The larger of two four-place airplanes");
      break;
    case "210":
      alert("A six-place high-performance airplane");
      break;
    default:
      alert("Error in JavaScript function planeChoice");
      break;
  }
}
```

```
// radio_click2r.js
//   The event registering code for radio_click2

var dom = document.getElementById("myForm");
dom.elements[0].onclick = planeChoice;
dom.elements[1].onclick = planeChoice;
dom.elements[2].onclick = planeChoice;
dom.elements[3].onclick = planeChoice;
```

• Assign the **address** of the handler function to the **event property** of the JavaScript object associated with the HTML element.

```
var dom = document.getElementById("myForm")
dom.elements[0].onclick = planeChoice;
```

• This registration must follow both the handler function and the HTML form

- • In this case, the `checked` property of a radio button object is used to determine whether a button is clicked

- An alternative approach to `radio_click2r.js` would be to give each radio button an id attribute and use the id to register the handler.

Handling Events from Text Box and Password Elements:

- Text boxes and passwords can create four different events: `blur`, `focus`, `change`, and `select`.
- Suppose JavaScript is used to compute the total cost of an order and display it to the customer before the order is submitted to the server for processing.
- For example:

```
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head> <title> nochange.html </title>

<!-- Script for the event handlers -->
    <script type = "text/javascript"  src = "nochange.js" >
    </script>

  </head>
  <body>
    <form action = "">
      <h3> Coffee Order Form </h3>

<!-- A bordered table for item orders -->
      <table border = "border">

<!-- First, the column headings -->
        <tr>
          <th> Product Name </th>
          <th> Price </th>
          <th> Quantity </th>
        </tr>


<!-- Now, the table data entries -->
        <tr>
          <th> French Vanilla (1 lb.) </th>
          <td> $3.49 </td>
          <td> <input type = "text"  id = "french"
                      size ="2" /> </td>
        </tr>
        <tr>
          <th> Hazlenut Cream (1 lb.) </th>
          <td> $3.95 </td>
          <td> <input type = "text"  id = "hazlenut"
                 size = "2" /> </td>
        </tr>
        <tr>
          <th> Colombian (1 lb.) </th>
          <td> $4.59 </td>
          <td> <input type = "text"  id = "colombian"
                 size = "2" /></td>
        </tr>
      </table>

<!-- Button for precomputation of the total cost -->
      <p>
        <input type = "button"  value = "Total Cost"
               onclick = "computeCost();" />
        <input type = "text"  size = "5"  id = "cost"
               onfocus = "this.blur();" />
      </p>
```

```
<!-- The submit and reset buttons -->
      <p>
        <input type = "submit"  value = "Submit Order" />
        <input type = "reset"  value = "Clear Order Form" />
      </p>
    </form>
  </body>
</html>
```

```
// nochange.js
//   This script illustrates using the focus event
//   to prevent the user from changing a text field

// The event handler function to compute the cost
function computeCost() {
  var french = document.getElementById("french").value;
  var hazlenut = document.getElementById("hazlenut").value;
  var colombian = document.getElementById("colombian").value;

// Compute the cost
  document.getElementById("cost").value =
  totalCost = french * 3.49 + hazlenut * 3.95 +
              colombian * 4.59;

}  //* end of computeCost
```

- In this example, the button labeled `Total Cost` allows the user to compute the total cost of the order before submitting the form.
- The event handler for this button gets the values (input quantities) of the three kinds of coffee and computes the total cost.
- The cost value is placed in the text box's value property, and it is then displayed for the user.
- Whenever this text box acquires focus, it is forced to blur with the `blur` method, which prevents the user from changing the value.

Validating Form Input:

- JavaScript can be used to prevent the form being submitted by checking the empty form field or input data validation.
- Data validation is the process of ensuring that user input is clean, correct, and useful.

Typical validation tasks are:

- has the user filled in all **required** fields?
- has the user entered a valid date?
- has the user entered text in a numeric field?
- Has the user entered correct email?,
- Match the two passwords etc.

- Validation can be defined by many different methods, and deployed in many different ways. The most common ways of data validation in HTML form are:
  - **Server side validation**: It is performed by a web server, after input has been sent to the server.
  - **Client side validation:** It is performed by a web browser, before input is sent to a web server. JavaScript is used for client-side validation.

Important Advantages and Disadvantages of Server and Client Side Validation:

Client-side Validation:

Advantages:

• Allow for more interactivity by immediately responding to users' actions at browser level.
• Execute quickly because they do not require a trip to the server that results in less network traffic.

Disadvantages:

- Client-side Validation can be easily bypassed by different techniques, so can never be most reliable technique.

Server-side Validation:

Advantages:

- Most often server side validation can **protect against the malicious user** from putting an invalid data which may corrupt database.
- Server-side validation is important for compatibility.

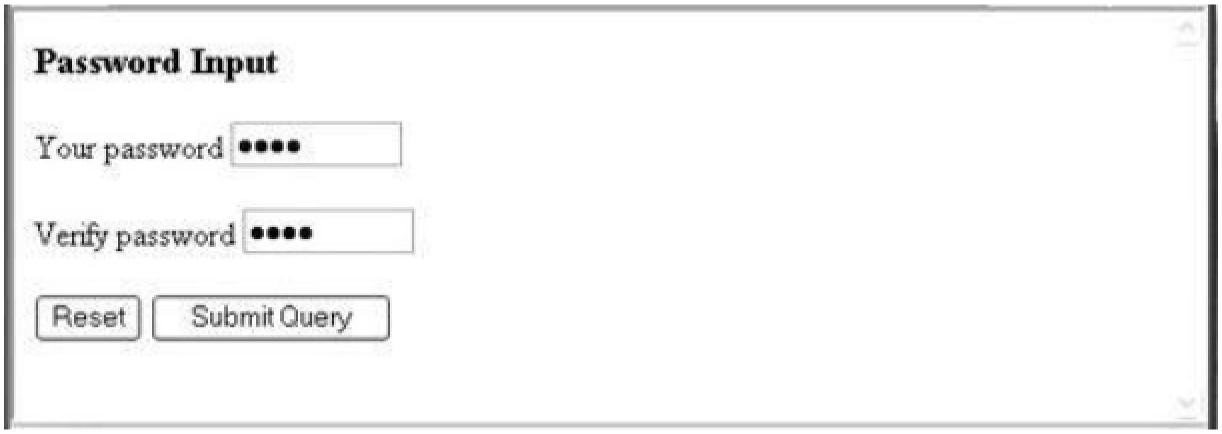
Note: Most often both client-side validation and server-side validation must be implemented for crucial data input. In some cases, Database validation can also be done if server-side validation fails.

- The following **Example 1** validates user entered passwords:

```
<?xml version = "1.0" encoding = "utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- pswd_chk.html
     A document for pswd_chk.ps
     Creates two text boxes for passwords
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Illustrate password checking> </title>
    <script type = "text/javascript"  src = "pswd_chk.js" >
    </script>
  </head>
  <body>
    <h3> Password Input </h3>
    <form id = "myForm"  action = "" >
      <p>

      <label> Your password
        <input type = "password" id = "initial"
               size = "10" />
      </label>
      <br /><br />

      <label> Verify password
        <input type = "password"  id = "second"
               size = "10" />
      </label>
      <br /><br />
      <input type = "reset"  name = "reset" />
      <input type = "submit"  name = "submit" />
      </p>
    </form>

<!-- Script for registering the event handlers  -->
    <script type = "text/javascript"  src = "pswd_chkr.js">
    </script>

  </body>
</html>
```

```
// pswd_chk.js
//   An example of input password checking using the submit
//   event

// The event handler function for password checking
function chkPasswords() {
  var init = document.getElementById("initial");
  var sec = document.getElementById("second");
  if (init.value == "") {
    alert("You did not enter a password \n" +
          "Please enter one now");
    return false;
  }
  if (init.value != sec.value) {
    alert("The two passwords you entered are not the same \n" +
          "Please re-enter both now");
    return false;
  } else
    return true;
}

// pswd_chkr.js
//   Register the event handlers for pswd_chk.html

document.getElementById("second").onblur = chkPasswords;
document.getElementById("myForm").onsubmit = chkPasswords;
```

OUTPUT:

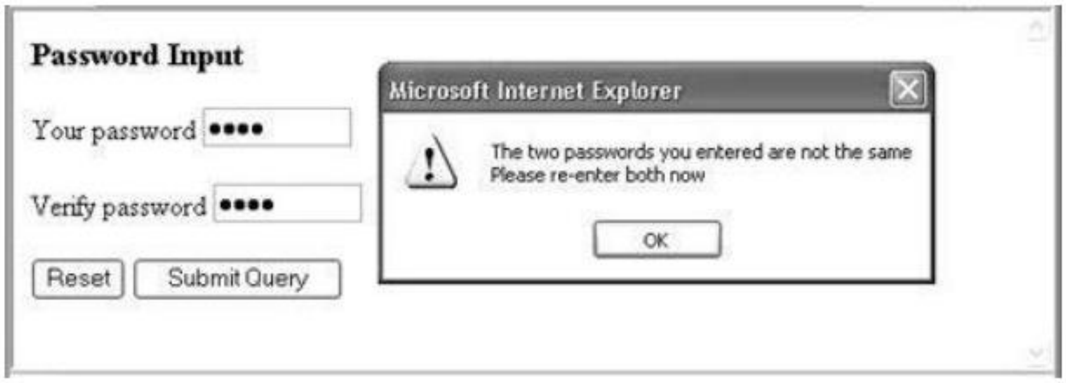


Password Input

Your password ••••

Verify password ••••

Fig. Display of pswd_chk.html after filled out



The two passwords you entered are not the same
Please re-enter both now

Fig. Display of pswd_chk.html after **Submit Query** has been clicked with unmatched passwords

- The following **Example 2** validates the name and phone number by pattern matching:

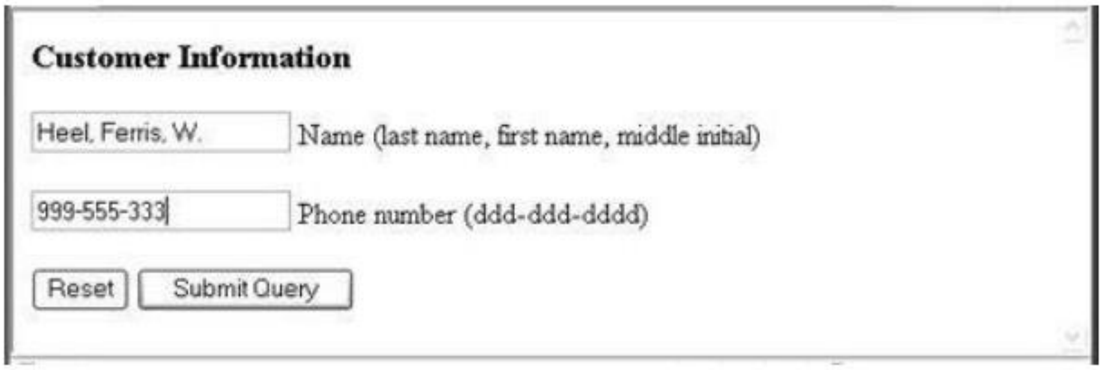
```
<!-- validator.html
     A document for validator.js
     Creates text boxes for a name and a phone number
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Illustrate form input validation> </title>
    <script type = "text/javascript"  src = "validator.js" >
    </script>
  </head>
  <body>
    <h3> Customer Information </h3>
    <form action = "">
      <p>
        <label>
          <input type = "text"  id = "custName" />
          Name (last name, first name, middle initial)
        </label>
        <br /><br />

        <label>
          <input type = "text"  id = "phone" />
          Phone number (ddd-ddd-dddd)
        </label>
        <br /><br />

        <input type = "reset"  id = "reset" />
        <input type = "submit"  id = "submit" />
      </p>
    </form>
    <script type = "text/javascript"  src = "validatorr.js">
    </script>
  </body>
</html>
```

Following script for event handlers and event registration:

```
// validator.js
//   An example of input validation using the change and submit
//   events

// The event handler function for the name text box
function chkName() {
  var myName = document.getElementById("custName");

// Test the format of the input name
//   Allow the spaces after the commas to be optional
//   Allow the period after the initial to be optional
  var pos = myName.value.search(
            /^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$/);
  if (pos != 0) {
    alert("The name you entered (" + myName.value +
          ") is not in the correct form. \n" +
          "The correct form is: " +
          "last-name, first-name, middle-initial \n" +
          "Please go back and fix your name");
    return false;
  } else
    return true;
}


// The event handler function for the phone number text box
function chkPhone() {
  var myPhone = document.getElementById("phone");

// Test the format of the input phone number
  var pos = myPhone.value.search(/^\d{3}-\d{3}-\d{4}$/);
  if (pos != 0) {
    alert("The phone number you entered (" + myPhone.value +
          ") is not in the correct form. \n" +
          "The correct form is: ddd-ddd-dddd \n" +
          "Please go back and fix your phone number");
    return false;
  } else
    return true;
}
```

```
// validatorr.js
//   Register the event handlers for validator.html

document.getElementById("custName").onchange = chkName;
document.getElementById("phone").onchange = chkPhone;
```

OUTPUTS:

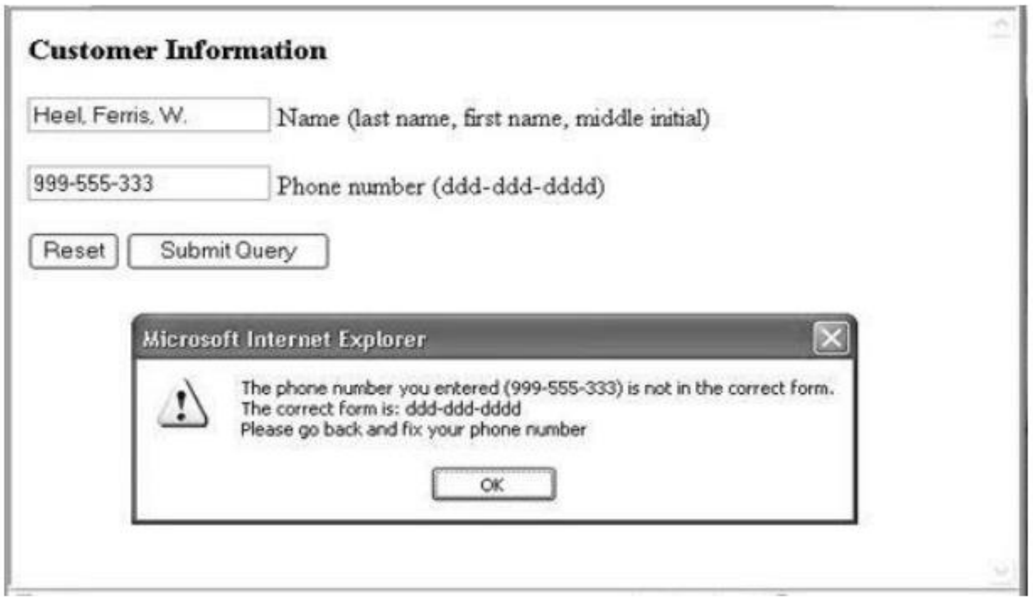


Customer Information

Heel, Ferris, W. Name (last name, first name, middle initial)

999-555-333 Phone number (ddd-ddd-dddd)

Fig. Display of **validator.html** with an invalid phone number while phone text-field focus



Customer Information

Heel, Ferris, W. Name (last name, first name, middle initial)

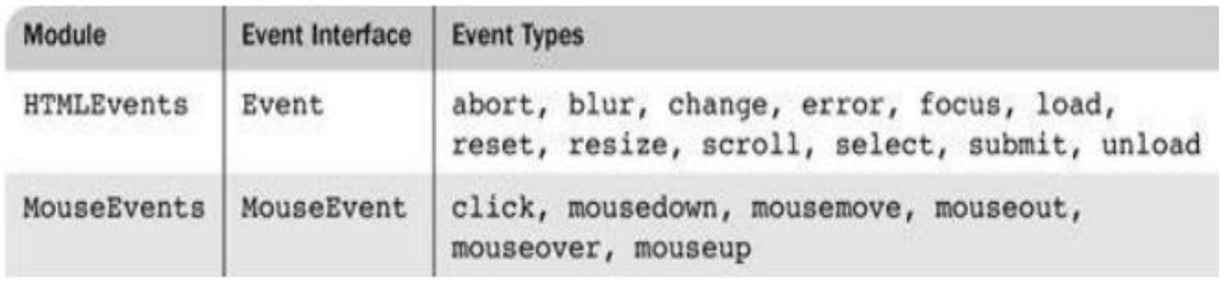
999-555-333 Phone number (ddd-ddd-dddd)

The phone number you entered (999-555-333) is not in the correct form.
The correct form is: ddd-ddd-dddd
Please go back and fix your phone number

Fig. The message created by entering an invalid phone number in **validator.html**

DOM 2 Event Model:

- Does not include DOM 0 features, but DOM 0 features are still supported by browsers
- The DOM 2 model is a modularized interface. One of the DOM 2 modules is **Events**, which includes several submodules. The ones most commonly used are **HTMLEvents** and **MouseEvents**. The interfaces and events defined by these modules are as follows:



Module	Event Interface	Event Types
`HTMLEvents`	`Event`	`abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload`
`MouseEvents`	`MouseEvent`	`click, mousedown, mousemove, mouseout, mouseover, mouseup`

Event Propagation:

- The node of the document tree where the event is created is called the *target node*
- The *capturing phase* (the first phase)
    - Events begin at the root and move toward the target node
    - Registered and enabled event handlers at nodes along the way are run
- The second phase is at the target node
    - If there are registered but not enabled handlers there for the event, they are executed
- The third phase is the *bubbling phase*
    - Event goes back to the root; all encountered registered but not enabled handlers are run

- Not all events bubble (e.g., `load` and `unload` events do not bubble. All of the mouse events bubble.)
- Any handler can stop further event propagation by calling the `stopPropagation` method of the `Event` object
- DOM 2 model uses the `Event` object method, `preventDefault`, to stop default operations, such as submission of a form, if an error has been detected
- Event handler registration is done with the `addEventListener` method:

  Three parameters:

  1. Name of the event, as a string literal
  2. The handler function
  3. A Boolean value that specifies whether the event is enabled during the capturing phase

```
node.addEventListener("change", chkName, false);
```

- A temporary handler can be created by registering it and then unregistering it with `removeEventListener`

- The `currentTarget` property of `Event` always references the object on which the handler is being executed.
- The MouseEvent interface (a subinterface of Event) has two properties, clientX and clientY, that have the x and y coordinates of the mouse cursor, relative to the upper left corner of the browser window.

For example:

```
<!DOCTYPE html>;
<!-- validator2.html
     A document for validator2.js
     Note: This document works with IE9, but not earlier
           versions of IE
     -->;
<html lang = "en">;
  <head>;
    <title>; Illustrate form input validation with DOM 2>; </title>;
    <meta charset = "utf-8" />;
    <script type = "text/javascript"  src = "validator2.js" >;
    </script>;
  </head>;
  <body>;
    <h3>; Customer Information </h3>;
    <form action = "">;
      <p>;
        <label>;
          <input type = "text"  id = "custName" />;
           Name (last name, first name, middle initial)
        </label>;
        <br />;<br />;

        <label>;
          <input type = "text"  id = "phone" />;
          Phone number (ddd-ddd-dddd)
        </label>;
        <br />;<br />;

        <input type = "reset" />;
        <input type = "submit"  id = "submitButton" />;
      </p>;
    </form>;
    <script type = "text/javascript"  src = "validator2r.js" >;
    </script>;

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

```
// validator2r.js
//   The last part of validator2. Registers the
//   event handlers
//   Note: This script does not work with IE8

// Get the DOM addresses of the elements and register
//  the event handlers

      var customerNode = document.getElementById("custName");
      var phoneNode = document.getElementById("phone");
      customerNode.addEventListener("change", chkName, false);
      phoneNode.addEventListener("change", chkPhone, false);

  // validator2.js
  //   An example of input validation using the change and submit
  //   events, using the DOM 2 event model
  //   Note: This document does not work with IE8

  // ********************************************************** //
  // The event handler function for the name text box

  function chkName(event) {

  // Get the target node of the event

    var myName = event.currentTarget;

  // Test the format of the input name
  //  Allow the spaces after the commas to be optional
  //  Allow the period after the initial to be optional

    var pos = myName.value.search(/^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-
  Z]\.?$/);

    if (pos != 0) {
      alert("The name you entered (" + myName.value +
            ") is not in the correct form. \n" +
            "The correct form is: " +
            "last-name, first-name, middle-initial \n" +
            "Please go back and fix your name");
     }
  }

  // ********************************************************** //
  // The event handler function for the phone number text box

  function chkPhone(event) {

  // Get the target node of the event

    var myPhone = event.currentTarget;

  // Test the format of the input phone number

    var pos = myPhone.value.search(/^\d{3}-\d{3}-\d{4}$/);
    if (pos != 0) {
      alert("The phone number you entered (" + myPhone.value +
            ") is not in the correct form. \n" +
            "The correct form is: ddd-ddd-dddd \n" +
            "Please go back and fix your phone number");
    }
  }
```

Difference between DOM Level 0 Events and DOM Level 2 Events:

DOM Level 0 events were based around the concept of using element attributes or named events on DOM elements, e.g.:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Traditional Event Handling</title>
</head>

<body>
  <h1>Traditional Event Handling</h1>

  <p>Hey Joe!</p>

  <script>
         var triggerAlert = function () {
                window.alert("Hey Joe");
         };

         // Assign an event handler
         document.onclick = triggerAlert;

         // Assign another event handler
         window.onload = triggerAlert;

         // Remove the event handler that was just assigned
         window.onload = null;
  </script>
</body>
</html>
```

With DOM Level 2, we've now got a more standardised approach to managing events and subscriptions, with `addEventListener`, `removeEventListener`, etc.

A rewrite of above example using DOM Level 2 event is as follows:

```
<!doctype html>
<html lang="en">
```

```
<head>
     <meta charset="utf-8">
     <title>DOM Level 2</title>
</head>

<body>
     <h1>DOM Level 2</h1>

     <p>Hey Joe!</p>


     <script>
          var heyJoe = function () {
               window.alert("Hey Joe!");
          }

          // Add an event handler
          document.addEventListener( "click", heyJoe, true );  //
capture phase

          // Add another event handler
          window.addEventListener( "load", heyJoe, false );  //
bubbling phase

          // Remove the event handler just added
          window.removeEventListener( "load", heyJoe, false );
     </script>

</body>
</html>
```

<u>**Positioning Elements:**</u>

- Mostly, the elements found in the HTML file were simply placed in the document the way text is placed in a document with a word processor: Fill a row, start a new row, fill it, and so forth.
- HTML tables provide a framework of columns for arranging elements, but they lack flexibility and also take a considerable time to display.
- Cascading Style Sheets–Positioning (CSS-P) provides the means not only to position any element anywhere in the display of a document, but also to move an element to a new position in the display dynamically, using JavaScript to change the positioning style properties of the element.
- The style properties, `left` and `top`, dictate the distance from the left and top of some reference point to where the element is to appear.
- Another style property, `position`, interacts with `left` and `top` to provide a higher level of control of placement and movement of elements.
- The `position` property has three possible values: `absolute`, `relative`, and `static`.

Absolute Positioning:

- The `absolute` value is specified for `position` when the element is to be placed at a specific place in the document display without regard to the positions of other elements.
- For example, if a paragraph of text is to appear 100 pixels from the left edge and 200 pixels from the top of the display window, the following element could be used:

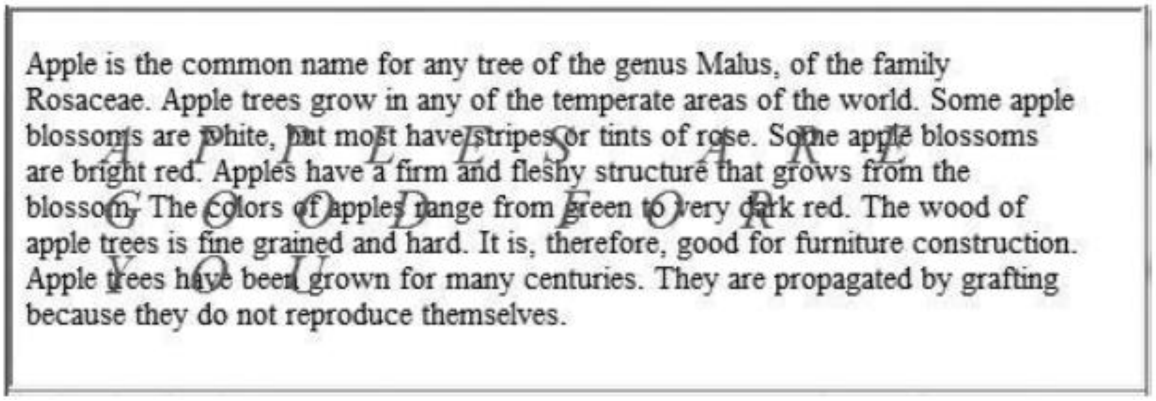
```
<p style = "position: absolute; left: 100px; top: 200px">
    -- text --
</p>
```

- One use of absolute positioning is to superimpose special text over a paragraph of ordinary text to create an effect similar to a watermark on paper.
- For example:

```
<!-- absPos.html
     Illustrates absolute positioning of elements
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Absolute positioning </title>
    <style type = "text/css">

/* A style for a paragraph of text */
     .regtext {font-family: Times; font-size: 14pt; width: 600px}

/* A style for the text to be absolutely positioned */
     .abstext {position: absolute; top: 25px; left: 50px;
               font-family: Times; font-size: 24pt;
               font-style: italic; letter-spacing: 1em;
               color: rgb(102,102,102); width: 500px}
    </style>
  </head>
  <body>

  <p class = "regtext">
    Apple is the common name for any tree of the genus Malus,
    of the family Rosaceae. Apple trees grow in any of the
    temperate areas of the world. Some apple blossoms are white,
    but most have stripes or tints of rose. Some apple blossoms
    are bright red. Apples have a firm and fleshy structure that
    grows from the blossom. The colors of apples range from
    green to very dark red. The wood of apple trees is fine
    grained and hard. It is, therefore, good for furniture
    construction. Apple trees have been grown for many
    centuries. They are propogated by grafting because they
    do not reproduce themselves.
  </p>
```

```
    <p class = "abstext">
      APPLES ARE GOOD FOR YOU
    </p>
  </body>
</html>
```

OUTPUT:



Apple is the common name for any tree of the genus Malus, of the family Rosaceae. Apple trees grow in any of the temperate areas of the world. Some apple blossoms are white, but most have stripes or tints of rose. Some apple blossoms are bright red. Apples have a firm and fleshy structure that grows from the blossom. The colors of apples range from green to very dark red. The wood of apple trees is fine grained and hard. It is, therefore, good for furniture construction. Apple trees have been grown for many centuries. They are propagated by grafting because they do not reproduce themselves.

A P P L E S A R E G O O D F O R Y O U

Fig. Display of absPos.html

Relative Positioning:

- An element that has the `position` property set to `relative`, but does not specify `top` and `left` property values, is placed in the document as if the `position` attribute were not set at all.
- For example, suppose that two buttons are placed in a document and the `position` attribute has its default value, which is `static`. Then the buttons would appear next to each other in a row, assuming that the current row has sufficient horizontal space for them.
- If `position` has been set to `relative` and the second button has its `left` property set to `50px`, the effect would be to move the second button 50 pixels farther to the right than it otherwise would have appeared.
- Relative positioning can be used for a variety of special effects in placing elements.
- For example, it can be used to create superscripts and subscripts by placing the values to be raised or lowered in `<span>` tags and displacing them from their regular positions.
- In the next example, a line of text is set in a normal font style in 24-point size. Embedded in the line is one word that is set in italic, 48-point, red font.
- Normally, the bottom of the special word would align with the bottom of the rest of the line.
- In this case, the special word is to be vertically centered in the line, so its `position` property is set to `relative` and its `top` property is set to 10 pixels, which lowers it by that amount relative to the surrounding text. The XHTML document to specify this, which is named `relPos.html`, is as follows:

```
<!-- relPos.html
     Illustrates relative positioning of elements
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Relative positioning </title>
  </head>
  <body style = "font-family: Times; font-size: 24pt;">
    <p>
      Apples are <span style =
              "position: relative; top: 10px;
               font-family: Times; font-size: 48pt;
               font-style: italic; color: red;">
      GOOD </span> for you.
    </p>
  </body>
</html>
```

OUTPUT:

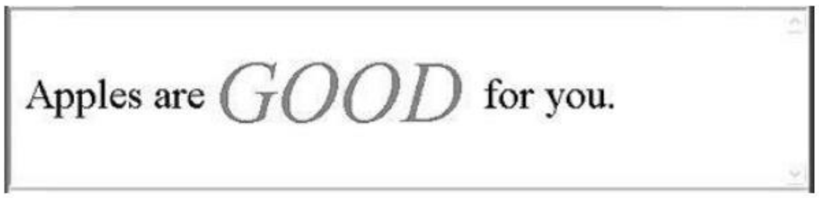

Apples are GOOD for you.

Fig. Display of relPos.html

Static Positioning:

- The default value for the `position` property is `static`. With `position: static`, the document moves along with the page. A statically positioned element is placed in the document as if it had the `position` value of `relative` but no values for `top` or `left` were given.
- The difference is that a statically positioned element cannot have its `top` or `left` properties initially set or changed later.
- Therefore, a statically placed element cannot be displaced from its normal position and cannot be moved from that position later.

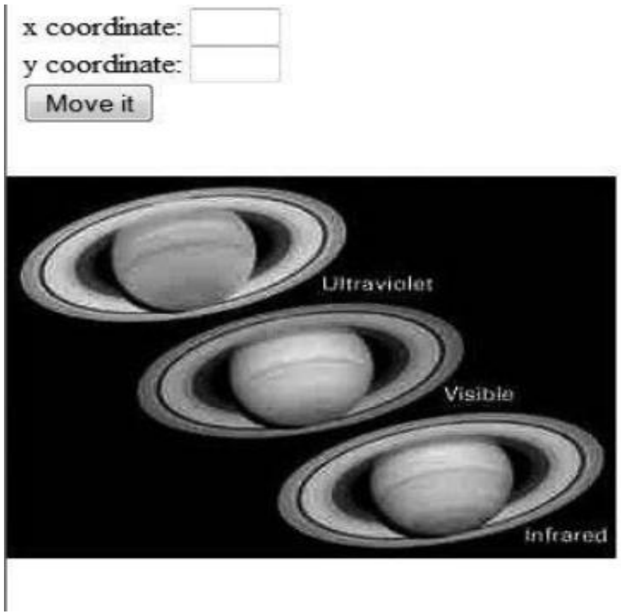
Moving Elements:

- An XHTML element whose `position` property is set to either `absolute` or `relative` can be moved.
- Moving an element is simple: Changing the `top` or `left` property values causes the element to move on the display.
- If its `position` is set to `absolute`, the element moves to the new values of `top` and `left`; if its `position` is set to `relative`, it moves from its original position by distances given by the new values of `top` and `left`.
- In the next example, an image is absolutely positioned in the display. The document includes two text boxes, labeled `x coordinate` and `y coordinate`, respectively.
- The user can enter new values for the `left` and `top` properties of the image in these boxes. When the button labeled Move It is pressed, the values of the `left` and `top` properties of the image are changed to the given values, and the element is moved to its new position.
- A JavaScript function, stored in a separate file, is used to change the values of `left` and `top` in this example.
- Although it is not necessary here, the id of the element to be moved is sent to the function that does the moving, just to illustrate that the function could be used on any number of different elements.
- The values of the two text boxes are also sent to the function as parameters.
- The actual parameter values are the DOM addresses of the text boxes, with the `value` attribute attached, which provides the complete DOM addresses of the text box values.
- Notice that `style` is attached to the DOM address of the image to be moved because `top` and `left` are style properties.
- Because the input `top` and `left` values from the text boxes are just string representations of numbers, but the `top` and `left` properties must end with some unit abbreviation, the event handler catenates `"px"` to each value before assigning it to the `top` and `left` properties.
- This document, called `mover.html`, and the associated JavaScript file, `mover.js`, are as follows:

```
<!-- mover.html
     Uses mover.js to move an image within a document
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Moving elements </title>
    <script type = "text/javascript"  src = "mover.js" >
    </script>
  </head>
  <body>
    <form action = "">
      <p>
        <label>
          x coordinate:
          <input type = "text"  id = "leftCoord" size = "3" />
        </label>
        <br />
        <label>
          y coordinate:
          <input type = "text"  id = "topCoord" size = "3" />
        </label>
        <br />


        <input type = "button"  value = "Move it"
               onclick =
                 "moveIt('saturn',
                 document.getElementById('topCoord').value,
                 document.getElementById('leftCoord').value)" />
      </p>
    </form>
    <div id = "saturn"  style = "position: absolute;
         top: 115px; left: 0;">
      <img src = "../images/ngc604.jpg"
           alt = "(Pictures of Saturn)" />
    </div>
  </body>
</html>
```

```
// mover.js
//   Illustrates moving an element within a document

// The event handler function to move an element
function moveIt(movee, newTop, newLeft) {
  dom = document.getElementById(movee).style;

// Change the top and left properties to perform the move
//  Note the addition of units to the input values
  dom.top = newTop + "px";
  dom.left = newLeft + "px";
}
```

OUTPUT:



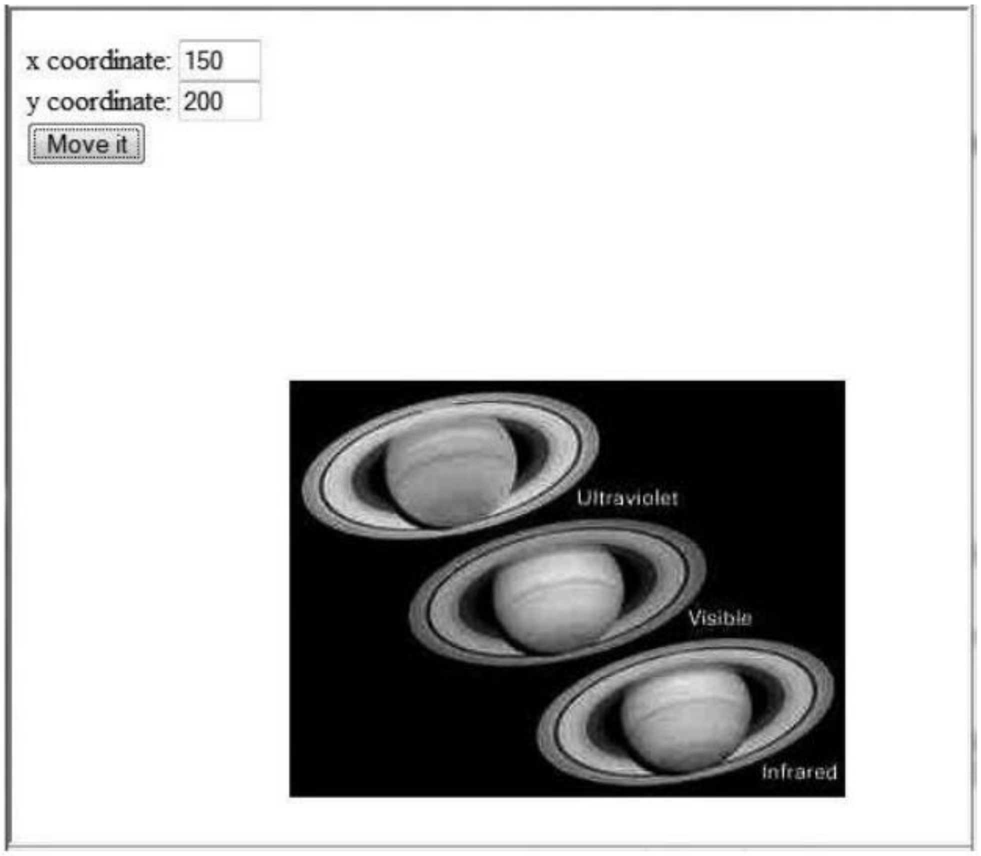


Fig. Display of mover.html (after pressing the *Move It* Button)

Element Visibility:

- Document elements can be specified to be visible or hidden with the value of their `visibility` property.
- The two possible values for `visibility` are, `visible` and `hidden`. The appearance or disappearance of an element can be controlled by the user through a widget.
- It is similar to the **display** property. However, the difference is that if you set `display:none`, it hides the entire element, while `visibility:hidden` means that the contents of the element will be invisible, but the element stays in its original position and size.
- For example:

```
<!DOCTYPE html>
<html>
<head>
<style>
#myDIV {
   width: 200px;
   height: 200px;
   background-color: lightblue;
}
</style>
</head>
<body>

<p>Click the "Try it" button to toggle between hiding and showing the DIV
element:</p>

<button onclick="myFunction()">Try it</button>

<div id="myDIV">
         This is my DIV element.
</div>

<p>Note that even though the element is hidden, it stays in its original position
and size.</p>

<script>
function myFunction() {
   var x = document.getElementById('myDIV');
   if (x.style.visibility === 'hidden') {
      x.style.visibility = 'visible';
   } else {
      x.style.visibility = 'hidden';
   }
}
</script>

</body>
</html>
```

Changing Fonts and Colors:

- The background, foreground colors and font properties of the text in a document display can be dynamically changed.

Changing Colors:

- The **color** property sets or returns the foreground color or color of the text while the **backgroundColor** property is used to set or return the background color of an element.

- The following example illustrates the changes in foreground and background color of textbox:

```
<!-- dynColors.html
     Uses dynColors.js
     Illustrates dynamic foreground and background colors
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Dynamic colors </title>
    <script type = "text/javascript"  src = "dynColors.js" >
    </script>
  </head>
  <body>
    <p style = "font-family: Times; font-style: italic;
                font-size: 24pt" >
      This small page illustrates dynamic setting of the
      foreground and background colors for a document
    </p>
    <form action = "">
      <p>
        <label>
          Background color:
          <input type = "text"  name = "background" size = "10"
                 onchange = "setColor('background', this.value)" />
        </label>
        <br />
        <label>
          Foreground color:
          <input type = "text"  name = "foreground" size = "10"
                 onchange = "setColor('foreground', this.value)" />
        </label>
        <br />
      </p>
    </form>
  </body>
</html>
```

```
// dynColors.js
//   Illustrates dynamic foreground and background colors

// The event handler function to dynamically set the
// color of background or foreground
function setColor(where, newColor) {
  if (where == "background")
    document.body.style.backgroundColor = newColor;
  else
    document.body.style.color = newColor;
}
```

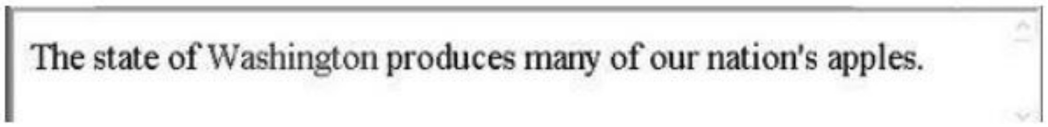
Changing Fonts:

- The **fontSize** property sets or returns the font size of the text.
- The following example illustrates the `fontSize, fontStyle and color` properties that change the `font-size, font-style and color` properties of word **Washington** in the sentence on `mouseover` and `mouseout` events.

```
<!-- dynFont.html
     Illustrates dynamic font styles and colors
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Dynamic fonts </title>
    <style type = "text/css">
      .regText {font: 16pt 'Times New Roman'}
      .wordText {color: blue;}
    </style>
  </head>
  <body>
    <p class = "regText">
```

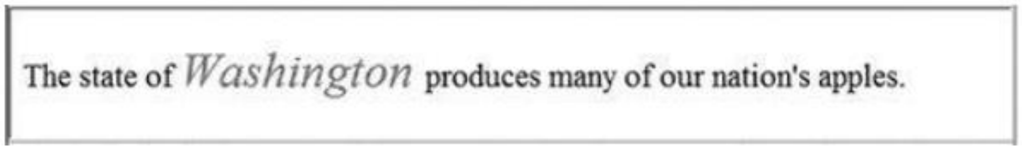
```
      The state of
      <span class = "wordText";
         onmouseover = "this.style.color = 'red';
                        this.style.fontStyle = 'italic';
                        this.style.fontSize = '24pt';"
         onmouseout = "this.style.color = 'blue';
                       this.style.fontStyle = 'normal';
                       this.style.fontSize = '16pt';">
         Washington
      </span>
      produces many of our nation's apples.
    </p>
  </body>
</html>
```

OUTPUT:



The state of Washington produces many of our nation's apples.

Fig. Display of `dynFont.html` with the mouse cursor not over the word.



The state of *Washington* produces many of our nation's apples.

Fig. Display of `dynFont.html` with mouse cursor over the word

Dynamic Content:

- JavaScript can be used to change dynamically the content of HTML/XHTML elements.
- The content of an element is accessed through the `value` property of its associated Java-Script object.
- So, changing the content of an element is not essentially different from changing other properties of the element.
- **Example 1:**

```
<!DOCTYPE html>
<html>
<body>
<p id="demo">JavaScript can change HTML content.</p>
<button type="button" onclick='document.getElementById("demo").innerHTML =
"Hello JavaScript!"'>Click Me!</button>
</body>
</html>
```

OUTPUT:

When button “Click Me!” is clicked the content of <p>, is changed to “Hello JavaScript!”

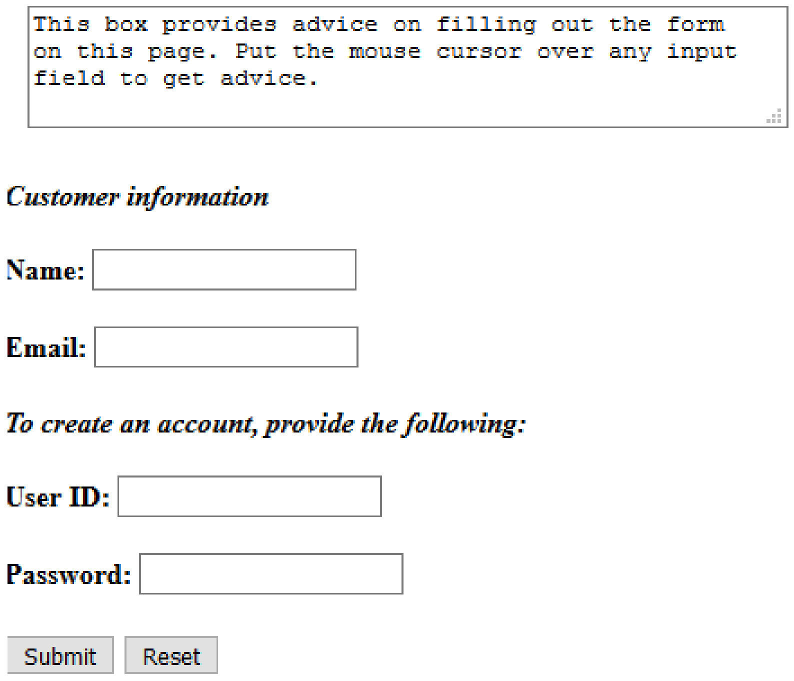
Example 2:

The following example shows the form that can be provided with an associated text area, called as help box. The content of the help box changes, depending on the placement of the mouse cursor. When the `mouseover` event is used i.e. cursor is placed over a particular input field, the help box can display advice on how the field is to be filled in. When the `mouseout` event is triggered or cursor is moved away from an input field, the help box content can be changed to simply indicate that assistance is available

```
       <!DOCTYPE html>
<!-- dynValue.html
     Illustrates dynamic values
     -->
<html lang = "en">
  <head>
    <title> Dynamic values </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "dynValue.js"
>
    </script>
    <style type = "text/css">
      textarea {position: absolute; left: 250px; top:
0px;}
      span {font-style: italic;}
      p {font-weight: bold;}
    </style>
  </head>
  <body>
    <form action = "">
      <p>
        <span>
          Customer information
        </span>
        <br /><br />
        <label>
          Name:
          <input type = "text"  onmouseover =
"messages(0)"
                 onmouseout = "messages(4)" />
        </label>
        <br />
        <label>
          Email:
          <input type = "text"  onmouseover =
"messages(1)"
                 onmouseout = "messages(4)" />
        </label>
        <br /> <br />
        <span>
          To create an account, provide the following:
        </span>
        <br /> <br />
        <label>
          User ID:
          <input type = "text"  onmouseover =
"messages(2)"
                 onmouseout = "messages(4)" />
        </label>
        <br />
        <label>
          Password:
          <input type = "password"
                 onmouseover = "messages(3)"
                 onmouseout = "messages(4)" />
        </label>
        <br />
      </p>
      <textarea id = "adviceBox"  rows = "3"  cols = "50">
        This box provides advice on filling out the form
        on this page. Put the mouse cursor over any input
        field to get advice.
      </textarea>
      <br /><br />
      <input type = "submit"  value = "Submit" />
      <input type = "reset"  value = "Reset" />
    </form>
  </body>
</html>
```

```
// dynValue.js
//   Illustrates dynamic values

var helpers = ["Your name must be in the form: \n \
 first name, middle initial., last name",
  "Your email address must have the form: \
 user@domain",
  "Your user ID must have at least six characters",
  "Your password must have at least six \
 characters and it must include one digit",
  "This box provides advice on filling out\
 the form on this page. Put the mouse cursor over any \
 input field to get advice"]

//
**********************************************************
*/
// The event handler function to change the value of the
//  textarea

function messages(adviceNumber) {
  document.getElementById("adviceBox").value =
                                helpers[adviceNumber];
}
```

OUTPUT:



This box provides advice on filling out the form on this page. Put the mouse cursor over any input field to get advice.

Customer information

Name:

Email:

To create an account, provide the following:

User ID:

Password:

Stacking Elements:

- The `top` and `left` properties allow the placement of an element anywhere in the two dimensions of the display of a document.
- Although the display is restricted to two physical dimensions, the effect of a third dimension is possible through the simple concept of stacked elements, n the document, one is considered to be on top and is displayed. The top element hides the parts of the lower elements on which it is superimposed. The placement of elements in this third dimension is controlled by the `z-index` attribute of the element.
- An element whose `z-index` is greater than that of an element in the same space will be displayed over the other element, effectively hiding the element with the smaller `z-index` value.
- The JavaScript style property associated with the `z-index` attribute is `zIndex`.

For example:

```
<!DOCTYPE html>

<html>

<head>

        <style>

        #myDIV {

           position: absolute;
```

```
            width: 300px;
            height: 150px;
            background-color: lightblue;
            border: 1px solid black;
        }
        #DIV2 {
            position: relative;
            top: 130px;
            left: 30px;
            width: 300px;
            height: 150px;
            background-color: coral;
            border: 1px solid black;
        }
        </style>
</head>
<body>
        <p>Click the "Try it" button to change the z-index of the blue DIV
element:</p>
        <button onclick="myFunction()">Try it</button>
<div id="DIV2">  <h1>Voila!</h1></div>
<div id="myDIV"></div>
        <script>
        function myFunction() {
            document.getElementById("myDIV").style.zIndex = "-1";
        }
        </script>
</body>
</html>
```

OUTPUT:

When user clicks "Try it" button then div with id="myDIV" is hidden and div with id=" DIV2".

Locating the Mouse Cursor:

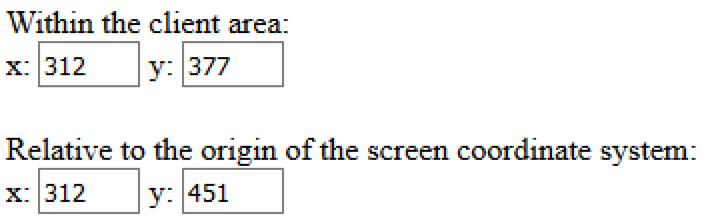
- The coordinates of the element that caused an event are available in the `clientX` and `clientY` properties of the `event` object
- These are relative to upper left corner of the browser display window
- `screenX` and `screenY` are relative to the upper left corner of the whole client screen
- If we want to locate the mouse cursor when the mouse button is clicked, we can use the `click` event

For example:

```
<!DOCTYPE html>
<!-- where.html
     Uses where.js
     Illustrates x and y coordinates of the mouse cursor
     -->
<html lang = "utf-8">
  <head>
    <title> Where is the cursor? </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "where.js" >
    </script>
  </head>
  <body onclick = "findIt(event)">
    <form action = "">
      <p>
        Within the client area: <br />
        x:
        <input type = "text"  id = "xcoor1"  size = "4" />
        y:
        <input type = "text"  id = "ycoor1"  size = "4" />
        <br /><br />
        Relative to the origin of the screen coordinate
system:
        <br />
        x:
        <input type = "text"  id = "xcoor2"  size = "4" />
        y:
        <input type = "text"  id = "ycoor2"  size = "4" />
      </p>
    </form>
    <p>
      <img src = "airplane1.jpg"  alt = "(Picture of an
airplane)" />
    </p>
  </body>
</html>
```

```
// where.js
//   Show the coordinates of the mouse cursor position
//   in an image and anywhere on the screen when the mouse
//   is clicked

// The event handler function to get and display the
//  coordinates of the cursor, both in an element and
//  on the screen

function findIt(evt) {
  document.getElementById("xcoor1").value = evt.clientX;
  document.getElementById("ycoor1").value = evt.clientY;
  document.getElementById("xcoor2").value = evt.screenX;
  document.getElementById("ycoor2").value = evt.screenY;
}
```

OUTPUT:



Within the client area:
x: 312 y: 377

Relative to the origin of the screen coordinate system:
x: 312 y: 451

Reacting to Mouse Click:

- A mouse click can be used to trigger an action, no matter where the mouse cursor is in the display
- The `mousedown` and `mouseup` events can be used, respectively, to trigger actions.
- For example:

```
<!DOCTYPE html>
<!-- anywhere.html
     Display a message when the mouse button is pressed,
     no matter where it is on the screen
     -->
<html lang = "en">
  <head>
    <title> Sense events anywhere </title>
    <meta charset = "utf-8" />
    <script type = "text/javascript"  src = "anywhere.js" >
    </script>
  </head>
  <body onmousedown = "displayIt(event);"
        onmouseup = "hideIt();">
    <p>
      <span id= "message"
            style = "color: red; visibility: hidden;
                     position: relative;
                     font-size: 1.7em; font-style: italic;
                     font-weight: bold;">
         Please don't click here!
      </span>
      <br /><br /><br /><br /><br /><br /><br /><br />
      <br /><br /><br /><br /><br /><br /><br /><br />
    </p>
  </body>
</html>
```

```
// anywhere.js
//   Display a message when the mouse button is pressed,
//   no matter where it is on the screen

// The event handler function to display the message

function displayIt(evt) {
  var dom = document.getElementById("message");
  dom.style.left = (evt.clientX - 130) + "px";
  dom.style.top = (evt.clientY - 25) + "px";
  dom.style.visibility = "visible";
}

// *****************************************************
// The event handler function to hide the message

function hideIt() {
  document.getElementById("message").style.visibility =
      "hidden";
}
```

OUTPUT:

The phrase "*Please don't click here!*" appears in reaction to the mouse click on the document.

Dragging and Dropping Elements:

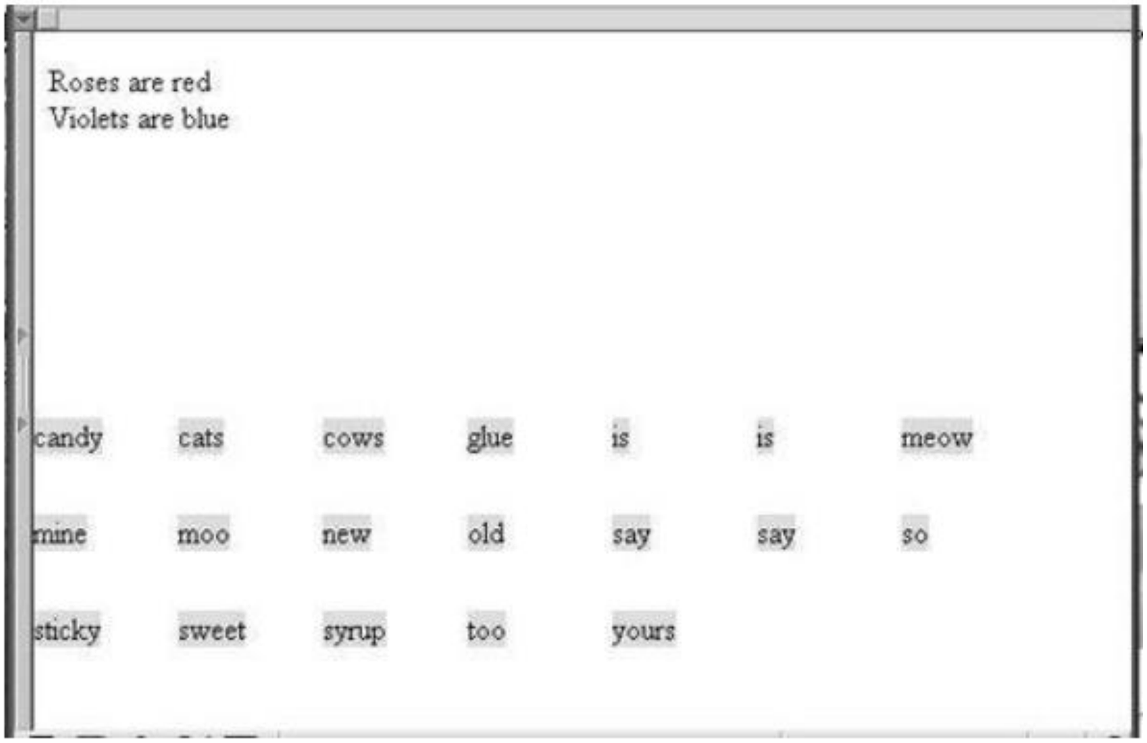
- The `mouseup`, `mousedown`, and `mousemove` events can be used to allow the user to drag and drop elements around the display screen.
- The clientX/Y, the pageX/Y and the screenX/Y methods of event object can be used for dragging and dropping of elements. To illustrate drag and drop, **the following example** that creates a magnetic poetry system is developed that completes lines of poem by dragging and dropping the words in required lines.

```
<!-- dragNDrop.html
     An example to illustrate the DOM 2 Event model
     Allows the user to drag and drop words to complete
     a short poem
     Does not work with IE8
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Drag and drop </title>
    <script type = "text/javascript"  src = "dragNdrop.js" >
    </script>
  </head>
  <body style = "font-size: 20;">
    <p>
      Roses are red <br />
      Violets are blue <br />

      <span style = "position: absolute; top: 200px; left: 0px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> candy </span>
      <span style = "position: absolute; top: 200px; left: 75px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> cats </span>
      <span style = "position: absolute; top: 200px; left: 150px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> cows </span>
      <span style = "position: absolute; top: 200px; left: 225px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> glue </span>
      <span style = "position: absolute; top: 200px; left: 300px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> is </span>
      <span style = "position: absolute; top: 200px; left: 375px;
                     background-color: lightgrey;"
            onmousedown = "grabber(event);"> is </span>
      <span style = "position: absolute; top: 200px; left: 450px;
                     background-color: lightgrey;"
```

```
       onmousedown = "grabber(event);"> meow </span>
 <span style = "position: absolute; top: 250px; left: 0px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> mine </span>
 <span style = "position: absolute; top: 250px; left: 75px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> moo </span>
 <span style = "position: absolute; top: 250px; left: 150px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> new </span>
 <span style = "position: absolute; top: 250px; left: 225px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> old </span>
 <span style = "position: absolute; top: 250px; left: 300px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> say </span>
 <span style = "position: absolute; top: 250px; left: 375px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> say </span>
 <span style = "position: absolute; top: 250px; left: 450px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> so </span>
 <span style = "position: absolute; top: 300px; left: 0px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> sticky </span>
 <span style = "position: absolute; top: 300px; left: 75px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> sweet </span>
 <span style = "position: absolute; top: 300px; left: 150px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> syrup </span>
 <span style = "position: absolute; top: 300px; left: 225px;
                background-color: lightgrey;"
       onmousedown = "grabber(event);"> too </span>
 <span style = "position: absolute; top: 300px; left: 300px;
                     background-color: lightgrey;"
           onmousedown = "grabber(event);"> yours </span>
   </p>
  </body>
</html>
```

```
// dragNDrop.js
//    An example to illustrate the DOM 2 Event model
//    Allows the user to drag and drop words to complete
//    a short poem
//    Does not work with IE8
// Define variables for the values computed by
// the grabber event handler but needed by mover
// event handler
      var diffX, diffY, theElement;

// *******************************************************
// The event handler function for grabbing the word
function grabber(event) {

// Set the global variable for the element to be moved
  theElement = event.currentTarget;

// Determine the position of the word to be grabbed,
// first removing the units from left and top
  var posX = parseInt(theElement.style.left);
  var posY = parseInt(theElement.style.top);

// Compute the difference between where it is and
// where the mouse click occurred
  diffX = event.clientX - posX;
  diffY = event.clientY - posY;

// Now register the event handlers for moving and
// dropping the word
  document.addEventListener("mousemove", mover, true);
  document.addEventListener("mouseup", dropper, true);



// Stop propagation of the event and stop any default
// browser action
  event.stopPropagation();
  event.preventDefault();

}  //** end of grabber

// *******************************************************
// The event handler function for moving the word
function mover(event) {

// Compute the new position, add the units, and move the word
  theElement.style.left = (event.clientX - diffX) + "px";
  theElement.style.top = (event.clientY - diffY) + "px";

// Prevent propagation of the event
  event.stopPropagation();
}  //** end of mover
// *********************************************************
// The event handler function for dropping the word
function dropper(event) {

// Unregister the event handlers for mouseup and mousemove
  document.removeEventListener("mouseup", dropper, true);
  document.removeEventListener("mousemove", mover, true);

// Prevent propagation of the event
  event.stopPropagation();
}  //** end of dropper
```

OUTPUT:



```
Roses are red
Violets are blue

candy    cats     cows     glue     is       is       meow

mine     moo      new      old      say      say      so

sticky   sweet    syrup    too      yours
```

Fig. Display of dragNDrop.html

Simple Code Synopsis for Dragging and Dropping elements:

HTML code:

```
<html>
<body>
        <head>
                .target{
                        height: 100px;
                        width: 100px;
                        background: #ccc;
                        position: absolute;
                        cursor: pointer;
                        }
        </head>
        <div class=”target”></div>
<script type=”text/javascript”>
        var ele = document.getElementsByClassName ("target")[0];
//ele.onmousedown = eleMouseDown;
ele.addEventListener ("mousedown" , eleMouseDown , false);

function eleMouseDown () {
  stateMouseDown = true;
  document.addEventListener ("mousemove" , eleMouseMove , false);
}

function eleMouseMove (ev) {
  var pX = ev.pageX;
  var pY = ev.pageY;
  ele.style.left = pX + "px";
  ele.style.top = pY + "px";
  document.addEventListener ("mouseup" , eleMouseUp , false);
}

function eleMouseUp () {
  document.removeEventListener ("mousemove" , eleMouseMove , false);
  document.removeEventListener ("mouseup" , eleMouseUp , false);
}
</script>
</body>
</html>
```