UNIT 2
JAVASCRIPT

OVERVIEW OF JAVASCRIPT

ORIGINS

- 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, this time to JavaScript.
- A language standard for JavaScript was developed in the late 1990s by the European Computer Manufacturers Association (ECMA) as ECMA-262.
- The official name of the standard language is ECMAScript.
- JavaScript can be divided into three parts: the core, client side, and server side.
- The ***core*** is the heart of the language, including its operators, expressions, statements, and subprograms.
- ***Client-side*** JavaScript is a collection of objects that support the control of a browser and interactions with users.
- ***Server-side*** JavaScript is a collection of objects that make the language useful on a Web server.

JAVASCRIPT AND JAVA

JAVA	JAVASCRIPT
Java is programming language	JavaScript is a scripting language
It is strongly typed language	It is dynamically typed language
Types are known at compile time	Compile time type checking is impossible
Objects in java are static	JavaScript objects are dynamic
Collection of data members and methods is fixed at compile time	The number of data members and methods of an object can change during execution
Object oriented programming language	Object based language

USES OF JAVASCRIPT

- The JavaScript was initially introduced to provide programming capability at both the server and client ends of web connection
- JavaScript therefore is implemented at 2 ends:
  - Client end
  - Server end
- The client side JavaScript is embedded in XHTML documents and is interpreted by the browser
- It also provides some means of computation, which serves as an alternative for some tasks done at the server side
- 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.
- For example, when a user moves the mouse cursor from a text box, JavaScript can detect that movement and check the appropriateness of the text box's value (which presumably was just filled by the user).
- Even without forms, user interactions are both possible and simple to program in JavaScript. These interactions, 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.

- This transfer of task ensures that the server is not overloaded and performs only required task
- But client side JavaScript cannot replace serves side JavaScript; because server side software supports file operations, database access, security, networking etc
- JavaScript is also used as an alternative to java applets.
- Programming in JavaScript is much simpler than compared to java
- JavaScript support DOM [Document Object Model] which enables JavaScript to access and modify CSS properties and content of any element of a displayed XHTML document

EVENT-DRIVEN COMPUTATION OF JAVASCRIPT

- In JavaScript, the actions are often executed in response to actions of the users of documents like mouse clicks and form submissions.
- This form of computation supports user interactions through the XHTML form elements on the client display
- One of the common uses of JavaScript is to check the values provided in forms by users to determine whether the values are sensible.
- The program or script on the server that processes the form data must check for invalid input data.
- When invalid data is found, the server must transmit that information back to the browser.
- Since this process is time consuming, we can perform input checks at the client side itself which saves both server time and internet time.
- However, validity checking is done on the server side because client side validity checking can be subverted by an unscrupulous user.

BROWSERS AND XHTML/JAVASCRIPT DOCUMENTS

★ If an XHTML document does not include embedded scripts, the browser reads the lines of the document and renders its window according to the tags, attributes, and content it finds.

★ When a JavaScript script is encountered in the document, the browser uses its JavaScript interpreter to "execute" the script.

★ Output from the script becomes the next markup to be rendered.

★ When the end of the script is reached, the browser goes back to reading the XHTML document and displaying its content.

★ There are two different ways to embed JavaScript in an XHTML document: implicitly and explicitly.

★ In explicit embedding, the JavaScript code physically resides in the XHTML document.

★ The JavaScript can be placed in its own file, separate from the XHTML document. This approach, called implicit embedding, has the advantage of hiding the script from the browser user.

★ When JavaScript scripts are explicitly embedded, they can appear in either part of an XHTML document—the head or the body—depending on the purpose of the script.

OBJECT ORIENTATION AND JAVASCRIPT

- JavaScript is an object-based language
- It supports 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

JAVASCRIPT OBJECTS

- In JavaScript, objects are collections of properties, which correspond to the members of classes in Java and C++.
- Each property is either a data property or a function or method property.
- Data properties appear in two categories: primitive values and references to other objects.

- JavaScript uses non-object types for some of its simplest types; these non-object types are called *primitives.*
- Primitives are used because they often can be implemented directly in hardware, resulting in faster operations on their values.
- All objects in a JavaScript program are indirectly accessed through variables.
- All primitive values in JavaScript are accessed directly—these are like the scalar types in Java and C++. These are often called *value types.*
- The properties of an object are referenced by attaching the name of the property to the variable that references the object.
- A JavaScript object appears, both internally and externally, as a list of property–value pairs.
- The properties are names; the values are data values or functions.
- All functions are objects and are referenced through variables.
- The collection of properties of a JavaScript object is dynamic: Properties can be added or deleted at any time.

GENERAL SYNTACTIC CHARACTERISTICS

- Scripts 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>
- Notice that the script element requires the closing tag, even though it has no content when the src attribute is included.
- In JavaScript, 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.
- JavaScript has 25 reserved words

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

- In addition, JavaScript has a large collection of predefined words, including alert, open, java, and self.
- JavaScript has two forms of comments, both of which are used in other languages. First, whenever two adjacent slashes (//) appear on a line, the rest of the line is considered a comment. 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, <!--.
- The following XHTML comment form hides the enclosed script from browsers that do not have JavaScript interpreters, but makes it visible to browsers that do support JavaScript:

- The use of semicolons in JavaScript is unusual. The JavaScript interpreter tries to make semicolons unnecessary, but it does not always work.
- When the end of a line coincides with what could be the end of a statement, the interpreter effectively inserts a semicolon there. But this can lead to problems. For example,

  return x;
- The interpreter will insert a semicolon after return, making x an invalid orphan.
- The safest way to organize JavaScript statements is to put each on its own line whenever possible and

terminate each statement with a semicolon. If a statement does not fit on a line, be careful to break the statement at a place that will ensure that the first line does not have the form of a complete statement.

PRIMITIVES, OPERATIONS, AND EXPRESSIONS

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.
- The difference between primitives and objects is shown in the following example.

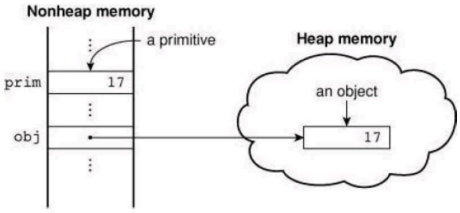


Figure 4.1 Primitives and objects

NUMERIC AND STRING LITERALS

- All numeric literals are values of type Number. The Number type values are represented internally in double-precision floating-point form.
- 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.
- The following are valid numeric literals:

  72 7.2 .72 72. 7E2 7e2 .7e2 7.e2 7.2E-2

- Integer literals can be written in hexadecimal form by preceding their first digit with either 0x or 0X.
- A string literal is a sequence of zero or more characters delimited by either single quotes (‘) or double quotes (“).
- String literals can include characters specified with escape sequences, such as \n and \t. If you want an actual single-quote character in a string literal that is delimited by single quotes, the embedded single quote must be preceded by a backslash:

  „You\"re the most lovely person I\"ve ever met"
- A double quote can be embedded in a double-quoted string literal by preceding it with a backslash. An actual backslash character in any string literal must be itself back-slashed, as in the following example:

  “D:\\bookfiles”
- There is no difference between single-quoted and double-quoted literal strings.
- The null string (a string with no characters) can be denoted with either ‘‘or “.

OTHER PRIMITIVE TYPES

- The only value of type Null is the reserved word null, which indicates no value.
- The only value of type Undefined is undefined.
- The only values of type Boolean are true and false.

DECLARING VARIABLES

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;
```

A variable that has been declared but not assigned a value, has the value undefined.

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.
- 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.
- All numeric operations are done in double-precision floating point.
- The precedence rules of a language specify which operator is evaluated first when two operators with different precedence are adjacent in an expression.
- The associativity rules of a language specify which operator is evaluated first when two operators with the same precedence are adjacent in an expression.

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

THE Math OBJECT

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.

THE NUMBER OBJECT

The Number object includes a collection of useful properties that have constant values. Table 4.3 lists the properties of Number. These properties are referenced through Number.

Property	Meaning
MAX_VALUE	Largest representable number
MIN_VALUE	Smallest representable number
NaN	Not a number
POSITIVE_INFINITY	Special value to represent infinity
NEGATIVE_INFINITY	Special value to represent negative infinity
PI	The value of π

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. 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. Example:

```
var price = 427,
str_price;
...
str_price = price.toString();
```

THE STRING CATENATION OPERATOR

JavaScript strings are not stored or treated as arrays of characters; rather, they are unit scalar values. String catenation is specified with the operator denoted by a plus sign (+). For example, if the value of first is “Divya”, the value of the following expression is “Divya Gowda”:

first + “ Gowda”

IMPLICIT TYPE CONVERSIONS

The JavaScript interpreter performs several different implicit type conversions. Such conversions are called ***coercions***. 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.

Example1: **“August ” + 1977 “August 1997”**

Example2: **7 * “3” 21** & will not be evaluated as string

EXPLICIT TYPE CONVERSIONS

Strings that contain numbers can be converted to numbers with the String constructor, as in the following code:

```
var str_value = String(value);
```

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.

```
var num = 6;
var str_value = num.toString();                    //the result is “6” var
str_value_binary = num.toString(2);                //the result is “110”
```

A number also can be converted to a string by catenating it with the empty string. Also,

```
var number = Number(aString);
```

The number in the string cannot be followed by any character except a space. JavaScript has two predefined string functions that do not have this problem.

- 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. 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. In both parseInt and parseFloat, the numeric literal could be followed by any nondigit character without causing any problem

String PROPERTIES AND METHODS

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;                    //now, len=6
```

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

Table 4.4 String methods

Consider, **var str = "George";**

Now, **str.charAt(2) is „o"**
str.indexOf(„r") is 3
str.substring(2, 4) is „org"
str.toLowerCase() is „george"

THE typeof OPERATOR

- The typeof operator returns the type of its single operand.
- 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.

ASSIGNMENT STATEMENTS

There is a simple assignment operator, denoted by =, and a host of compound assignment operators, such as += and /=. For example, the statement **a += 7;** means the same as **a = a + 7;**

THE Date OBJECT

A Date object is created with the new operator and the Date constructor, which has several forms.

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).

Table 4.5 shows the methods, along with the descriptions, that retrieve information from a 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

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 modelled 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.
- write is used to create XHTML code, the only useful punctuation in its parameter is in the form of XHTML tags. Therefore, the parameter of write often includes
.
- The writeln method implicitly adds "\n" to its parameter, but since browsers ignore line breaks when displaying XHTML, it has no effect on the output.
- The parameter of write can include any XHTML tags and content.
- The write method actually can take any number of parameters.
- Multiple parameters are concatenated and placed in the output.
- Example: **document.write("The result is: ", result, "
");**

- There are 3 types of pop-up boxes:
  - Alert
  - Confirm
  - Prompt
- The alert method opens a dialog window and displays its parameter in that window. It also displays an OK button.
- The string parameter of alert is not XHTML code; it is plain text. Therefore, the string parameter of alert may include \n but never should include
.

alert("The sum is:" + sum + "\n");

- 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. This method is often used to offer the user the choice of continuing some process.

var question = confirm("Do you want to continue this download?");

- After the user presses one of the buttons in the confirm dialog window, the script can test the variable, question, and react accordingly.

- 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.

CONTROL STATEMENTS

A compound statement in JavaScript is a sequence of statements delimited by braces.
A control construct is a control statement together with the statement or compound statement whose execution it controls.

CONTROL 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). A relational expression has two operands and one relational operator. Table 4.6 lists the 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	!==

JavaScript has operators for the AND, OR, and NOT Boolean operations. These are && (AND), || (OR), and ! (NOT). Both && and || are short-circuit operators.

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

Table 4.7 Operator precedence and associativity
Highest-precedence operators are listed first.

SELECTION STATEMENTS

The selection statements (if-then and if-then-else) of JavaScript are similar to those of the common programming languages. Either single statements or compound statements can be selected—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 />");
}
```

THE switch STATEMENT

JavaScript has a switch statement that is similar to that of Java. In any case segment, the statement(s) can be either a sequence of statements or a compound statement. The break statement transfers control out of the compound statement in which it appears.

The control expression of a switch statement could evaluate to a number, a string, or a Boolean value. Case labels also can be numbers, strings, or Booleans, and different case values can be of different types.

Example:

//switch.js

```
var choice = prompt("Select your favorite star \n" + "1
                    Puneeth Rajkumar \n" +
                    "2 Mahesh Babu \n" +
                    "3 Suriya \n");

switch(choice)
{
        case "1": document.write("Puneeth");
                  break;
        case "2": document.write("Mahesh Babu");
                  break;
        case "3": document.write("Suriya");
                  break;
        default: document.write("invalid choice");
}
```

//switch.html

```
<html>
<body>
<script type = "text/javascript" src = "switch.js">
</script>
</body>
</html>
```

Assignment: switch statement for table border size selection – it is similar but refer text book

LOOP STATEMENTS

The general form of the while statement is as follows:

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

The general form of the for statement is as follows:

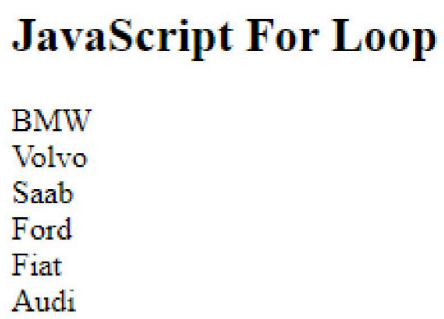
```
var i;
for (i = 0; i < cars.length; i++) {
  text += cars[i] + "<br>";
}
```

The following example illustrates the simple for loop:

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

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

<script>
var cars = ["BMW", "Volvo", "Saab", "Ford", "Fiat", "Audi"];
var text = "";
```

```
    var i;
    for (i = 0; i < cars.length; i++) {
      text += cars[i] + "<br>";
    }
    document.getElementById("demo").innerHTML = text;
    </script>

    </body>
    </html>
```

OUTPUT:



JavaScript has a do-while statement, whose form is as follows:

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

JavaScript includes one more loop statement, the for-in statement, which is most often used with objects.

```
for (identifier in object)
  statement or compound statement
```

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, however, the new operator creates a blank object—that is, one with no 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. Because properties are not variables, they are never declared.
- The number of members of a class in a typical object-oriented language is fixed at compile time. 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. A property for an object is created by assigning a value to that property's name. Consider the following example:
  var my_car = {make: "Ford", model: "Contour SVT"};
- Properties can be accessed in two ways.

var prop1 = my_car.make; var prop2 = my_car["make"];

the variables prop1 and prop2 both have the value "Ford".

- A property can be deleted with delete, as in the following example:

delete my_car.model;

- JavaScript has a loop statement, for-in, that is perfect for listing the properties of an object.

```
for (var prop in my_car)
  document.write("Name: ", prop, "; Value: ",
                 my_car[prop], "<br />");
```

ARRAYS

Array OBJECT CREATION

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);
```

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"];

CHARACTERISTICS OF Array OBJECTS

The lowest index of every JavaScript array is zero. Access to the elements of an array is specified with numeric subscript expressions placed in brackets. The length of an array is the highest subscript to which a value has been assigned, plus 1.

For example, if my_list is an array with four elements and the following statement is executed, the new length of my_list will be 48.

my_list[47] = 2222;

The length of an array is both read and write accessible through the length property, which is created for every array object by the Array constructor. For example,

my_list.length = 1002;

An array is lengthened by setting its length property to a larger value, shortened by setting its length property to a smaller value.

Array METHODS

Array objects have a collection of useful methods, most of which are described in this section.

- 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. If a string parameter is provided, it is used as the element separator. Consider the following 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
- The **concat** method catenates its actual parameters to the end of the Array object on which it is called.

```
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 does for arrays what the substring method does for strings, returning the part of the Array object specified by its parameters, which are used as subscripts. The array returned has the elements of the Array object through which it is called, from the first parameter up to, but not including, the second parameter.

```
var list = [2, 4, 6, 8, 10];
...
var list2 = list.slice(1, 3);
```

The value of list2 is now [4, 6]. If slice is given just one parameter, the array that is returned has all of the elements of the object, starting with the specified index.

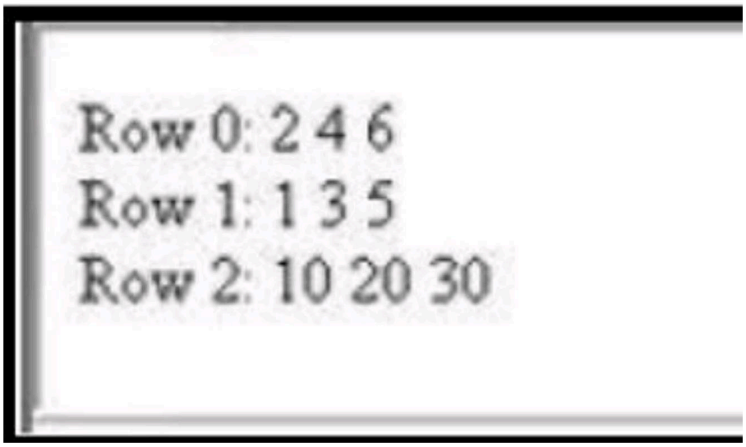
- When the **toString** method is called through an Array object, each of the elements of the object is converted (if necessary) to a string. These strings are catenated, separated by commas. So, for Array objects, the toString method behaves much like join.
- The **push, pop, unshift,** and **shift** 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.
var deer = list.shift(); // deer is now "Dasher" list.unshift("Dasher"); // This puts "Dasher" back on list

```
// nested_arrays.js
//   An example illustrating an array of arrays

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

// Display the elements of nested_list
for (var row = 0; row <= 2; row++) {
  document.write("Row ", row, ":  ");

  for (var col = 0; col <=2; col++)
    document.write(nested_array[row][col], " ");

  document.write("<br />");
}
```

OUTPUT:



FUNCTIONS

FUNDAMENTALS

- A function definition consists of the function's header and a compound statement that describes the actions of the function. This compound statement is called the body of the function.
- A function header consists of the reserved word function, the function's name, and a parenthesized list of parameters if there are any.
- A return statement returns control from the function in which it appears to the function's caller. A function body may include one or more return statements. If there are no return statements in a function or if the specific return that is executed does not include an expression, the value returned is undefined.
- JavaScript functions are objects, so variables that reference them can be treated as are other object references—they can be passed as parameters, be assigned to other variables, and be the elements of an array. The following example is illustrative:

```
function fun() { document.write(
                 "This surely is fun! <br/>");}

ref_fun = fun;   // Now, ref_fun refers to the fun object
fun();           // A call to fun
ref_fun();       // Also a call to fun
```

- Because JavaScript functions are objects, their references can be properties in other objects, in which case they act as methods.

LOCAL VARIABLES

- The scope of a variable is the range of statements over which it is visible.
- When JavaScript is embedded in an XHTML document, the scope of a variable is the range of lines of the document over which the variable is visible.
- Variables that are implicitly declared have global scope—that is, they are visible in the entire XHTML document.
- It is usually best for variables that are used only within a function to have local scope, meaning that they are visible and can be used only within the body of the function. Any variable explicitly declared with var in the body of a function has local scope.
- If a variable that is defined both as a local variable and as a global variable appears in a function, the local variable has precedence, effectively hiding the global variable with the same name. This is the advantage of local variables.

PARAMETERS

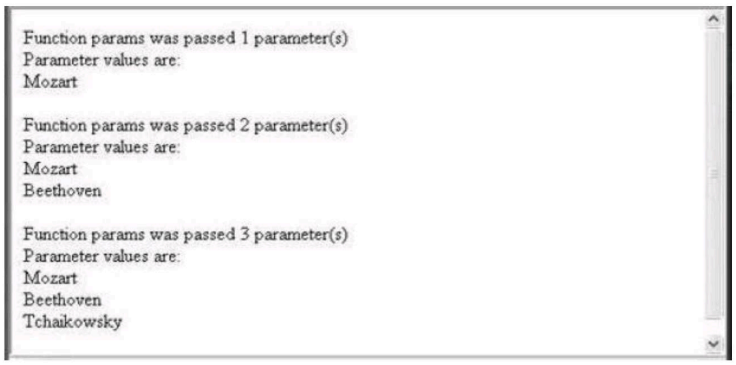
- The parameter values that appear in a call to a function are called actual parameters.
- The parameter names that appear in the header of a function definition, which correspond to the actual parameters in calls to the function, are called formal parameters.

- JavaScript uses the pass-by-value parameter-passing method.
- When a function is called, the values of the actual parameters specified in the call are, in effect, copied into their corresponding formal parameters, which behave exactly like local variables.
- Because of JavaScript's dynamic typing, there is no type checking of parameters. The called function itself can check the types of parameters with the typeof operator.
- The number of parameters in a function call is not checked against the number of formal parameters in the called function.
- In the function, excess actual parameters that are passed are ignored; excess formal parameters are set to undefined.
- All parameters are communicated through a property array, arguments, that, like other array objects, has a property named length.
- By accessing arguments.length, a function can determine the number of actual parameters that were passed.

- The following example illustrates a variable number of function parameters:

```
// params.js
//   The params function and a test driver for it.
//   This example illustrates a variable number of
//   function parameters

// Function params
// Parameters: A variable number of parameters
// Returns: nothing
// Displays its parameters
function params(a, b) {
  document.write("Function params was passed ",
      arguments.length, " parameter(s) <br />");
  document.write("Parameter values are: <br />");

  for (var arg = 0; arg < arguments.length; arg++)
    document.write(arguments[arg], "<br />");

  document.write("<br />");
}

// A test driver for function params
params("Mozart");
params("Mozart", "Beethoven");
params("Mozart", "Beethoven", "Tchaikowsky");
```

Output:



There is no elegant way in JavaScript to pass a primitive value by reference. One inelegant way is to put the value in an array and pass the array, as in the following script:

```
// Function by10
//    Parameter: a number, passed as the first element
//               of an array
//  Returns: nothing
//  Effect: multiplies the parameter by 10
function by10(a) {
    a[0] *= 10;
}
...
var x;
var listx = new Array(1);
...
listx[0] = x;
by10(listx);
x = listx[0];
```

THE sort METHOD, REVISITED

- If you need to sort something other than strings, or if you want an array to be sorted in some order other than alphabetically as strings, the comparison operation must be supplied to the sort method by the caller. Such a comparison operation is passed as a parameter to sort.
- The comparison function must return a negative number if the two elements being compared are in the desired order, zero if they are equal, and a number greater than zero if they must be interchanged.
- For example, if you want to use the sort method to sort the array of numbers num_list into descending order, you could do so with the following code:

```
// Function num_order
// Parameter: Two numbers
// Returns: If the first parameter belongs before the
//           second in descending order, a negative number
//         If the two parameters are equal, 0
//         If the two parameters must be
//           interchanged, a positive number
function num_order(a, b) {return b - a;}
// Sort the array of numbers, list, into
// ascending order
  num_list.sort(num_order);
```

AN EXAMPLE

```
// medians.js
//   A function and a function tester
//   Illustrates array operations

// Function median
//   Parameter: An array of numbers
//   Result: The median of the array
//   Return value: none
function median(list) {
  list.sort(function (a, b) {return a - b;});
  var list_len = list.length;

// Use the modulus operator to determine whether
//   the array's length is odd or even
// Use Math.floor to truncate numbers
// Use Math.round to round numbers
  if ((list_len % 2) == 1)
    return list[Math.floor(list_len / 2)];
  else
    return Math.round((list[list_len / 2 - 1] +
                       list[list_len / 2]) / 2);
}  // end of function median
var my_list_1 = [8, 3, 9, 1, 4, 7];
var my_list_2 = [10, -2, 0, 5, 3, 1, 7];
var med = median(my_list_1);
document.write("Median of [", my_list_1, "] is: ",
               med, "<br />");
med = median(my_list_2);
document.write("Median of [", my_list_2, "] is: ",
               med, "<br />");
```

Output:

CONSTRUCTORS

- 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.
- Constructors are actually called by the new operator, which immediately precedes them in the new expression.
- 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");

- For example, suppose you wanted a method for car objects that listed the property values. A function that could serve as such a method could be written as follows:

```
function display_car() {
    document.write("Car make: ", this.make, "<br/>");
    document.write("Car model: ", this.model, "<br/>");
    document.write("Car year: ", this.year, "<br/>");
}
```

- The following line must then be added to the car constructor:

this.display = display_car;

- Now the call my_car.display() will produce the following output

```
Car make: Ford
Car model: Contour SVT
Car year: 2000
```

PATTERN MATCHING BY 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.
- 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.
- Most characters are normal, which means that, in a pattern, they match themselves.
- The position of the first character in the string is 0.
- As an example, the following statements

```
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

- 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.
- A period matches any character except newline.
- Example: **/snow./** matches “snowy”, “snowe”, and “snowd”
- Example: **/3\.4/** matches 3.4. *but* **/3.4/** would match 3.4 and 374, among others.
- Example: **[abc]** matches ‘a’ , ‘b’ & ‘c’
- Example: **[a-h]** matches any lowercase letter from ‘a’ to ‘h’
- Example: **[^aeiou]** matches any lowercase letter except ‘a’, ‘e’, ‘i’, ‘o’ & ‘u’

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

ANCHORS

- A pattern is tied to a string position with an anchor. A pattern can be specified to match only at the beginning of the string by preceding it with a circumflex (^) anchor.
- For example, the following pattern matches “pearls are pretty” but does not match “My pearls are pretty”:
  /^pearl/
- A pattern can be specified to match at the end of a string only by following the pattern with a dollar sign anchor. For example, the following pattern matches “I like gold” but does not match “golden”:
  /gold$/
- Anchor characters are like boundary-named patterns: They do not match specific characters in the string; rather, they match positions before, between, or after characters.

PATTERN MODIFIERS

- The modifiers are specified as letters just after the right delimiter of the pattern.
- The i modifier makes the letters in the pattern match either uppercase or lowercase letters in the string.
- For example, the pattern /Apple/i matches ‘APPLE’, ‘apple’, ‘APPle’, and any other combination of uppercase and lowercase spellings of the word “apple.”

- The x modifier allows white space to appear in the pattern.

```
/\d+          # The street number
\s            # The space before the street name
[A-Z][a-z]+   # The street name
/x
```

is equivalent to

```
/\d+\s[A-Z][a-z]+/
```

OTHER PATTERN-MATCHING METHODS OF String

- 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.
- The g modifier can be attached to the pattern if the replacement is to be global in the string, in which case the replacement is done for every match in the string.
- The matched substrings of the string are made available through the predefined variables $1, $2, and so on. For example, consider the following statements:
  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".
- The match method is the most general of the String pattern-matching methods.
- The match method takes a single parameter: a pattern. It returns an array of the results of the pattern-matching operation.
- If the pattern has the g modifier, the returned array has all of the substrings of the string that matched.
- If the pattern does not include the g modifier, the returned array has the match as its first element, and the remainder of the array has the matches of parenthesized parts of the pattern if there are any:

```
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].
- The split method of String splits its object string into substrings on the basis of a given string or pattern. The substrings are returned in an array. For example, consider the following code:
  var str = "grapes:apples:oranges"; var fruit = str.split(":");
  In this example, fruit is set to [grapes, apples, oranges].

ANOTHER EXAMPLE

```
// forms_check.js
//   A function tst_phone_num is defined and tested.
//   This function checks the validity of phone
//   number input from a form

// Function tst_phone_num
//   Parameter: A string
//   Result: Returns true if the parameter has the form of a valid
//           seven-digit phone number (3 digits, a dash, 4 digits)
function tst_phone_num(num) {

// Use a simple pattern to check the number of digits and the dash
  var ok = num.search(/^\d{3}-\d{4}$/);

  if (ok == 0)
    return true;
  else
    return false;

}  // end of function tst_phone_num
```

```
// A script to test tst_phone_num
var tst = tst_phone_num("444-5432");
if (tst)
  document.write("444-5432 is a valid phone number <br />");
else
  document.write("Error in tst_phone_num <br />");
tst = tst_phone_num("444-r432");
if (tst)
  document.write("Program error <br />");
else
  document.write(
            "444-r432 is not a valid phone number <br />");

tst = tst_phone_num("44-1234");
if (tst)
  document.write("Program error <br />");
else
  document.write("44-1234 is not a valid phone number <br /");
```

OUTPUT:

444-5432 is a valid phone number
444-r432 is not a valid phone number
44-1234 is not a valid phone number

ERRORS IN SCRIPTS

The JavaScript interpreter is capable of detecting various errors in scripts. Debugging a script is a bit different from debugging a program in a more typical programming language, mostly because errors that are detected by the JavaScript interpreter are found while the browser is attempting to display a document. In some cases, a script error causes the browser not to display the document and does not produce an error message. Without a diagnostic message, you must simply examine the code to find the problem.

```
//pswd_chk.html
<html>
<head>
<title>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="final" size="10"/></label><br/><br/>
<input type="reset" name="Reset"/>
<input type="submit" name="Submit"/>
</p>
</form>
<script type = "text/javascript" src = "pswd_chkr.js">
 </script>
</body>
</html>
```

```
//pswd_chk.js
function chkPass( )
{
var init=document.getElementById("initial");
var fin=document.getElementById("final");
if(init.value=="")
{
alert("You did not enter a Password\n" + "Please enter atleast now");
init.focus( );
return false;
}

if(init.value!=fin.value)
{
alert("The passwords you entered do not match. Try Again");
init.focus( );
init.select( );
return false;
}
Else return true;
}
```

We now consider an example that checks the validity of the form values for a name and phone number obtained from text boxes. The pattern for matching names [LastName, FirstName, MiddleName] is as follows:

/^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$/

The pattern for phone numbers is as follows:

/^d{3}-\d{8}$/

The following is the XHTML document, validator.html, that displays the text boxes for a customer's name and phone number:

```
//validator.html
<html>
<head>
<title>Name and Phone check</title>
<script type = "text/javascript" src = "validator.js">
</script>
</head>
<body>
<h3>enter your details</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="custPhone"/>Phone (ddd-ddddddddd)</label><br/><br>
<input type="reset" id="reset"/>
<input type="submit" id="submit"/>
</p>
</form>
<script type = "text/javascript">
document.getElementById("custName").onchange=chkName;
document.getElementById("custPhone").onchange=chkPhone;
</script>
</body>
</html>

//validator.js
function chkName()
{
        var myName = document.getElementById("custName");
        var pos = myName.value.search(/^[A-Z][a-z]+, ?[A-Z][a-z]+, ?[A-Z]\.?$/);
        if(pos != 0)
        {
                alert("The name you entered (" + myName.value + ") is not in the correct form.\n" + "The
                        correct form is: " + "last-name, first-name, middle-initial \n" +
                            "Please go and fix your name");
                myName.focus();
                myName.select();
                return false;
        }
        else return true;
}
function chkPhone()
{
        var myPhone = document.getElementById("custPhone");
        var pos = myPhone.value.search(/^\d{3}-\d{8}$/); if(pos !=
        0)
        {
                alert("The phone you entered (" + myPhone.value + ") is not in the correct form.\n" +
                        "The correct form is: " + "ddd-ddddddddd \n" +
                            "Please go and fix your phone number");
                myPhone.focus();
                myPhone.select();
                return false;
        }
        else return true;
}
```

THE DOM 2 EVENT MODEL

The DOM 2 model is a modularized interface. One of the DOM 2 modules is Events, which includes several sub-modules. 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

// nochange.html

```
<html>
 <head><title>nochange.html</title>
  <script type = "text/javascript" src = "nochange.js">
  </script>
 </head>
 <body>
  <form action = " ">
   <h3> Non-Veg Items Order Form</h3>
   <table border="border">
<tr>
<th>Item</th>
<th>Price</th>
<th>Quantity</th>
</tr>
<tr>
<th>Chicken Kabab (full)</th>
<td>Rs. 150</td>
<td><input type = "text" id = "chicken" size = "2"/></td>
</tr>
<tr>
<th>Mutton Kaima (half)</th>
<td>Rs. 250</td>
<td><input type = "text" id = "mutton" size = "2"/></td>
</tr>
<tr>
<th>Fish Fry (2 pieces)</th>
<td>Rs. 100</td>
<td><input type = "text" id = "fish" size = "2"/></td>
</tr>
</table>
<p>
<input type = "button" value = "Total Cost" onclick =
"computeCost();"/>
<input type = "text" size = "5" id = "cost" onfocus =
"this.blur();"/>
</p>
<p>
<input type = "submit" value = "Submit Order"/>
<input type = "reset" value = "Clear Order Form"/>
</p>
</form>
</body>
</html>
```

/nochange.js

```
unction computeCost()
{
var chicken = document.getElementById("chicken").value;
var mutton = document.getElementById("mutton").value; var
ish = document.getElementById("fish").value;

document.getElementById("cost").value = totalCost = chicken*150
+ mutton*250 + fish*100;
}
```

Output:

After taking the entries, we get,

EVENT PROPAGATION:

- A browser which understands DOM, on receiving the XHTML document from the server, creates a tree known as document tree.
- The tree constructed consists of elements of the document except the HTML
- The root of the document tree is document object itself
- The other elements will form the node of the tree
- In case of DOM2, the node which generates an event is known as target node
- Once the event is generated, it starts the propagation from root node
- During the propagation, if there are any event handlers on any node and if it is enabled then event handler is executed
- The event further propagates and reaches the target node.
- When the event handler reaches the target node, the event handler gets executed
- After this execution, the event is again re-propagated in backward direction
- During this propagation, if there are any event handlers which are enabled, will be executed.
- The propagation of the even from the root node towards the leaf node or the target node is known as
  capturing phase.
- The execution of the event handler on the target node is known as **execution phase.**
- This phase is similar to event handling mechanism in DOM – 0
- The propagation of the event from the leaf or from the target node is known as **bubbling phase**
- All events cannot be bubbled for ex: load and unload event
- If user wants to stop the propagation of an event, then stop propagation has to be executed.

EVENT REGISTRATION:

- In case of DOM2, the events get registered using an API known as **addEventListener**
- The first arg is the eventName. Ex: click, change, blur, focus
- The second arg is the event handler function that has to be executed when there is an event
- The third arg is a Boolean argument that can either take a true or false value
- If the value is true, it means event handler is enabled in capturing phase
- If the event value if off (false), then event handler is enabled at target node
- The addEventListener method will return event object to eventhandler function. The event object can be accessed using the keyword “Event”
- The address of the node that generated event will be stored in **current target**, which is property of
  event object

AN EXAMPLE OF THE DOM 2 EVENT MODEL

The next example is a revision of the validator.html document and validator.js script from previous example, which used the DOM 0 event model. Because this version uses the DOM 2 event model, it does not work with IE8.

//validator2.html

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

```
<title>Illustrate form input validation with DOM 2</title>
<script type = "text/javascript" src = "validator2.js">
</script>
</head>
<body>
<h3>enter your details</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="custPhone"/>Phone (ddd-ddddddddd)</label><br/><br>
<input type="reset" />
<input type="submit" id="submitButton"/>
</p>
</form>
<script type = "text/javascript" src = "validator2r.js"/>
</body>
</html>
```

//validator2.js

```
function chkName(event)
{
var myName = event.currentTarget;
var pos = myName.value.search(/^[A-Z][a-z]+, ?[A-Z][a-
z]+, ?[A-Z]\.?$/); if(pos != 0)
{
alert("The name you entered (" + myName.value + ") is not in the
        correct form.\n" + "The correct form is: " + "last-name, first-
        name, middle-initial \n" +
            "Please go
and fix your name");
myName.focus();
myName.select();
}
}
function chkPhone(event)
{
var myPhone = event.currentTarget;
var pos =
myPhone.value.search(/^\d{3}-
\d{8}$/); if(pos != 0)
{
alert("The phone you entered (" + myPhone.value + ") is not in the
        correct form.\n" + "The correct form is: " + "ddd-ddddddddd \n"
        +
            "Please go and fix your
phone number");
myPhone.focus();
myPhone.select();
}
}
```

//validator2r.js

```
var c =
document.getElementById("cust
Name"); var p =
document.getElementById("cust
Phone");
c.addEventListener("change",chk
Name,false);
p.addEventListener("change",chk
Phone,false);
```

THE navigator OBJECT

The navigator object indicates which browser is being used to view the XHTML document. The browser's name is stored in the appName property of the object. The version of the browser is stored in the appVersion property of the object. These properties allow the script to determine which browser is being used and to use

processes appropriate to that browser. The following example illustrates the use of navigator, in this case just to display the browser name and version number:

```
//navigate.html file
<html>
<head>
<title>Navigator</title>
<script type = "text/javascript" src = "navigate.js">
</script>
</head>
<body onload = "navProperties()">
</body>
</html>
//navi
gate.js
file
functi
on
navPr
operti
es()
{
alert("the browser is: " +
navigator.appName + "\n" + "the
version number is: " +
navigator.appVersion + "\n");
}
```

DOM TREE TRAVERSAL AND MODIFICATION

DOM TREE TRAVERSAL

The parentNode property has the DOM address of the parent node of the node through which it is referenced. The childNodes property is an array of the child nodes of the node through which it is referenced. The previousSibling property has the DOM address of the previous sibling node of the node through which it is referenced. The nextSibling property has the DOM address of the next sibling node of the node through which it is referenced. The firstChild and lastChild properties have the DOM addresses of the first and last child nodes, respectively, of the node through which they are referenced. The nodeType property has the type of the node through which it is referenced.

DOM TREE MODIFICATION

A number of methods allow JavaScript code to modify an existing DOM tree structure. The insertBefore(newChild, refChild) method places the newChild node before the refChild node. The replaceChild(newChild, oldChild) method replaces the oldChild node with the newChild node. The removeChild(oldChild) method removes the specified node from the DOM structure. The appendChild(newChild) method adds the given node to the end of the list of siblings of the node through which it is called.