Prepared by: Bilal Zahoor
Kashmir University, North Campus

Introduction to JavaScript

JavaScript (JS) is a **lightweight**, **interpreted**, programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as **node.js** and Apache CouchDB. JS is a **prototype-based**, **multi-paradigm**, **dynamic scripting language**, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

JavaScript runs on the client side of the web, which can be used to design / program how the web pages behave on the occurrence of an event. JavaScript is an easy to learn and also powerful scripting language, widely used for controlling web page behavior. The basic syntax is intentionally similar to both Java and C++ to reduce the number of new concepts required to learn the language.

A Brief History of JavaScript

JavaScript was developed by Brendan Eich, a developer at Netscape Communications Corporation, in 1995. Its initial development was very rapid, and much of the criticism leveled at JavaScript has cited the lack of planning foresight during its development.
However, Brendan Eich was not a dabbler: he had a solid foundation in computer science, and incorporated remarkably sophisticated and prescient ideas into JavaScript. In many ways, it was ahead of its time, and it took 15 years for mainstream developers to catch on to the sophistication the language offered.

JavaScript started life with the name **Mocha**, and was briefly named **Live Script** before being officially renamed to JavaScript in a Netscape Navigator release in 1995. The word “Java” in “JavaScript” was not coincidental, but it is confusing: aside from a common syntactic ancestry, JavaScript has more in common with **Self** (a proto type based language developed at Xerox PARC in the mid-’80s) and **Scheme** (a language developed in the 1970s by Guy Steele and Gerald Sussman, which was in turn heavily influenced by Lisp and ALGOL) than with Java. Eich was familiar with both **Self** and **Scheme**, and used some of their forward-thinking paradigms in developing JavaScript. The name JavaScript was partially a marketing attempt to tie into the success Java was enjoying at the time.

In November 1996, Netscape announced that they had submitted JavaScript to Ecma, a private, international nonprofit standards organization that carries significant influence in the technology and communications industries. Ecma International published the first edition of the ECMA-26 specification, which was, in essence, JavaScript.

The standard for JavaScript is ECMAScript. As of 2012, all modern browsers fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3. On June 17, 2015, ECMA International published the sixth major version of ECMAScript, which is officially called ECMAScript 2015, and is more commonly referred to as ECMAScript 6 or ES6.

Client-Side JavaScript

A *scripting language* is a language used to manipulate, customize, or automate the facilities of an existing system. In the case of JavaScript, that system is typically the Web browser and its associated technologies of HTML, CSS, and XML. JavaScript itself is a relatively simple language, and much of its power is derived from both the built-in and document objects provided by the browser.
JavaScript has quickly become the premier client-side scripting language used within Web pages. Much of the language‘s success has to do with the ease with which developers can start using it. While JavaScript is quite powerful as a client-side technology, like all languages, it is better at some types of applications than others. Some of these common uses of JavaScript include

- Form validation
- Page embellishments and special effects
- Navigation systems
- Basic mathematical calculations
- Dynamic document generation
- Manipulation of structured documents

Server-Side JavaScript

Server-side JavaScript (SSJS) refers to JavaScript that runs on server-side and is therefore not downloaded to the browser. This term is used to differentiate it from regular JavaScript, which is predominantly used on the client-side (also referred to as client-side JavaScript or CSJS for short). The first implementation of SSJS was Netscape's **LiveWire**, which was included in their Enterprise Server 2.0 back in 1996. Since then, a number of other companies have followed suit in offering an alternative to the usual server-side technologies. They supported the use of JavaScript on the server within what is now known as "classic" **ASP (Active Server Pages).**

Node.js is also server side JavaScript**.** It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

JavaScript Objects

JavaScript is an object-based language. With the exception of language constructs like loops and relational operators, almost all of JavaScript‘s features are implemented using objects in one way or another.

Objects in JavaScript fall into four groups:

1) **User-defined objects** are custom objects created by the programmer to bring structure and consistency to a particular programming task.

2) **Built-in objects** are provided by the JavaScript language itself. These include those objects associated with data types (String, Number, and Boolean), objects that allow creation of user-defined objects and composite types (Object and Array), and objects that simplify common tasks, such as Date, Math, and RegExp.
3) **Browser objects** are those objects not specified as part of the JavaScript language but that most browsers commonly support. Examples of browser objects include Window, the object that enables the manipulation of browser windows and interaction with the user, and Navigator, the object that provides information about client configuration. Because most aspects of browser objects are not governed by any standard, their properties and behavior can vary significantly from browser to browser and from version to version.
4) **Document objects** are part of the Document Object Model (DOM), as defined by the W3C. These objects present the programmer with a structured interface to (X)HTML and XML documents. It is these objects that enable JavaScript to manipulate Cascading Style Sheets (CSS) and that facilitate the realization of Dynamic HTML (DHTML). Access to the document objects is provided by the browser via the document property of the Window object (window.document).

JavaScript Security

The modern JavaScript security model is based upon Java. In theory, downloaded scripts are run by default in a restricted ”**sandbox**” environment that isolates them from the rest of the operating system. Scripts are permitted access only to data in the current document or closely related documents (generally those from the same site as the current document). No access is granted to the local file system, the memory space of other running programs, or the operating system‘s networking layer. Containment of this kind is designed to prevent malfunctioning or malicious scripts from wreaking havoc in the user‘s environment. The reality of the situation, however, is that often scripts are not contained as neatly as one would hope. There are numerous ways that a script can exercise power beyond what you might expect, both by design and by accident. The fundamental premise of browsers‘ security models is that there is no reason to trust randomly encountered code such as that found on Web pages, so JavaScript should be executed as if it were hostile.

Operators in JavaScript

JavaScript supports a variety of operators. Some of them, like those for arithmetic and comparison, are easy for even those new to programming to understand. Others, like the bitwise AND (&), increment (**++**), and some conditional (**?:**) operators, may be less obvious to those who have not programmed before. Fortunately for readers of all levels, JavaScript supports few operators that are unique to the language, and the language mimics C, C++, and Java closely in both the kinds of operators it provides and their functionality.

Arithmetic Operators

JavaScript supports all the basic arithmetic operators that readers should be familiar with, including addition (**+**), subtraction (**–**), multiplication (*****), division (**/**), and modulus (**%**, also known as the remainder operator). Table given below all these operators and presents examples of each.

Operator	**Description**	**Example**

+	Addition (also string concatenation)	var x = 5, y = 7;
		var sum;
		sum = x+y; //12

-	Subtraction	var x = 5, y = 7;
		var diff1, diff2;
		diff1 = x–y; // -2
		diff2 = y–x; // 2

/	Division	var x = 36, y = 9, z = 5;
		var div1, div2;
		div1 = x / y; // 4
		div2 = x / z; // 7.2

*	Multiplication	var x = 8, y = 4;
		var product;
		product = x*y; //32

%	Remainder	var x = 24, y = 5, z = 6;
		var mod1, mod2;
		mod1 = x%y; // 4
		mod2 = x%z; // 0

-	Unary negation	-x // negative x; if x is 5, -x will be -5
+	Unary plus	+x // if x is not a number, this will attempt conversion
++	Pre-increment	++x // increments x by one, and evaluates to the new Value
++	Post-increment	x++ // evaluates to value of x before the increment
--	Pre-decrement	--x // decrements x by one, and evaluates to the new Value
--	Post-decrement	x-- // evaluates to value of x before the decrement

Comparison Operators

A comparison expression evaluates to a Boolean value indicating whether its comparison is **true** or **false**. Comparison operators, as the name implies, are used to compare two different values.

Broadly speaking, there are three types of comparison operator: **strict equality**, **abstract** (or loose) equality, and relational. Two values are considered strictly equal if they refer to the same object, or if they are the same type and have the same value (for primitive types). The advantage of strict equality is that the rules are very simple and straightforward, making it less prone to bugs and misunderstandings. To determine if values are strictly equal, use the === operator or its opposite, the not strictly equal operator (!==). Two values are considered abstractly equal if they refer to the same object or *if they can be coerced into having the same value*. For example, if you want to know if the number 33 and the string "33" are equal, the abstract equality operator will says yes, but the strict equality operator will say no (because they aren't the same type).

Table given below all these operators and presents examples of each.

Operator	**Description**	**Example**	**Evaluates**
<	Less than	4 < 8	True
<=	Less than or equal to	6 <= 5	False
>	Greater than	4 > 3	True
> =	Greater than or equal to	5 > = 5	True
!=	Not equal to	6 != 5	True
==	Equal to	6 == 5	False
===	Equal to (and have same type)	6 === '6'	False
!==	Not equal to (or don't have same type)	5 !== '5'	True

Logical Operators

Logical operators operate only on boolean values and return only boolean values. JavaScript allows you to operate on values that are not boolean, and even more surprising, can return values that aren't boolean. That is not to say JavaScript's implementation of logical operators is somehow wrong or not rigorous: if you use only boolean values, you will get results that are only boolean values.

Truthy and Falsy Values

Many languages have a concept of "truthy" and "falsy" values; C, for example, doesn't even have a boolean type: numeric 0 is false, and all other numeric values are true.

JavaScript does something similar, except it includes all data types, effectively allowing you to partition any value into truthy or falsy buckets. JavaScript considers the following values to be falsy:

- Undefined
- Null
- False
- 0
- NaN
- '' (an empty string)

Everything else is truthy. Because there are a great number of things that are truthy, some you should be conscious of:

- Any object (including an object whose valueOf() method returns false)
- Any array (even an empty array)
- Strings containing only whitespace (such as " ")
- The string "false"

One notable exception might be the fact that an empty array is truthy. If you want an array to be falsy if it's empty, use arr.length (which will be 0 if the array is empty, which is falsy).

The three logical operators supported by JavaScript are AND (&&), OR (||), and NOT (!).A description and example of each logical operator are shown in table:

Operator	Description	Example

&&	Returns **true** if both operands evaluate **true;** otherwise returns **false**.	var x=true, y=false
		alert(x && y);
		// displays false

||	Returns **true** if either operand is **true**. If both are **false**, returns **false**.	var x=true, y= false
		alert(x || y);
		// displays true

!	If its single operand is **true**, returns **false;** otherwise returns **true**.	var x=true
		alert(!x);
		// displays false

Short-Circuit Evaluation

X && Y is false whenever X is false, you don't even need to consider the value of Y. Similarly, if you're evaluating X || Y, and X is true, you don't need to evaluate Y. JavaScript does exactly this, and it's known as ***short-circuit evaluation***.

Why is short-circuit evaluation important? Because if the second operand has *side effects*, they will not happen if the evaluation is short-circuited.

In an expression, side effects can arise from incrementing, decrementing, assignment, and function calls. Let's see an example:

```
var skipIt = true;
var x = 0;
var result = skipIt || x++;
```

The second line in that example has a direct result, stored in the variable **result**. That value will be true because the first operand (skipIt) is true. What's interesting, though, is because of the short-circuit evaluation, the increment expression isn't evaluated, leaving the value of x at 0. If you change skipIt to false, then both parts of the expression have to be evaluated, so the increment will execute: the increment is the side effect. The same thing happens in reverse with AND:

```
var doIt = false;
var x = 0;
var result = doIt && x++;
```

Again, JavaScript will not evaluate the second operand, which contains the increment, because the first operand to AND is false. So result will be false, and x will not be incremented.

Conditional Operator (?: Operator)

The conditional operator is JavaScript's sole *ternary* operator, meaning it takes three operands (all other operands take one or two). The conditional operator is the expression equivalent of an **if...else** statement. One major difference between **?:** and **if** is that the **?:** operator allows only a single statement for the **true** and **false** conditions. The basic syntax for this operator is:

(*expression*) ? *if-true-statement* : *if-false-statement;*

where *expression* is any expression that will evaluate eventually to **true** or **false**. If *expression* evaluates **true**, *if-true-statement* is evaluated. Otherwise, *if-false-statement* is executed.
In this example,

```
(x >> 5) ? alert("x is greater than 5") : alert("x is less than 5");
```

An alert dialog will be displayed based upon the value of the variable *x*. Contextually, if the conditional expression evaluates **true**, the first statement indicating the value is greater than 5 is displayed; if **false**, the second statement stating the opposite will be displayed.

Comma Operator (,)

The comma operator provides a simple way to combine expressions: it simply evaluates two expressions and returns the result of the second one. It is convenient when you want to execute more than one expression, but the only value you care about is the result of the final expression.
Here is a simple example:

```
var x = 0, y = 10, z;
```

```
z = (x++, y++);
```

In this example, x and y are both incremented, and z gets the value 10 (which is what y++ returns). Note that the comma operator has the lowest precedence of any operator, which is why we enclosed it in parentheses here: if we had not, z would have received 0 (the value of x++), and *then* y would have been incremented. You most commonly see this used to combine expressions in a **for loop** or to combine multiple operations before returning from a function.

Void Operator

The **void** operator specifies an expression to be evaluated without returning a value. The void operator has only one job: to evaluate its operand and then return undefined. For example, take the previous example with the comma operator and void it out:

```
var a,b,c,d;
a = void (b=5, c=7, d=56);
document.write('a = '+a+' b = '+b+' c = '+c+' d = ' + d);
```

In this case, the value of *a* will be **undefined**,

The most common use of the **void** operator is when using the **javascript:** pseudo-URL in conjunction with an HTML **href** attribute. Some browsers, notably early versions of Netscape, had problems when script was used in links. The only way to avoid these problems and force a link click to do nothing when scripting is on is to use **void**, as shown here:

```
<a href="javascript:void (alert('hi!'))">  Click me! </a>
```

typeof

The **typeof** operator returns a string indicating the data type of its operand. The script fragment here shows its basic use:

```
 a = 3;
name = "Howard";
alert(typeof a);              // displays number
alert(typeof name);        // displays string
```

The typeof operator has one quirk that's usually referred to as a bug: **typeof null** returns "object". null is not, of course, an object (it is a primitive). The reasons are historical and not particularly interesting, and it has been suggested many times that it be fixed, but too much existing code is already built on this behavior, so it's now immortalized in the language specification.

Table shows the values returned by **typeof** on the basis of the type of value it is presented.

Expression	Return Values	Notes
typeof undefined	“undefined”	
typeof null	“object”	Unfortunate, but true
typeof {}	“object”	
typeof true (Boolean)	“boolean”	
typeof 2 (Number)	“number”	
type “ ” (String)	“string”	
typeof Symbol()	“symbol”	New in ES6
typeof function() {}	“function”	

Assignment Operators
The assignment operator is straightforward: it assigns a value to a variable. What’s on the left hand side of the equals sign (sometimes called the *lvalue*) *must* be a variable, property, or array element. That is, it must be something that can hold a value (assigning a value to a constant is technically part of the declaration, not an assignment operator). Assignment is itself an expression, so the value that’s returned is the value that’s being assigned. This allows you to chain assignments, as well as perform assignments within other expressions:

```
var v, v1;
v = v1 = 9.8;       // chained assignment; first v1 gets
                    // the value 9.8, and then v gets the
                    // value 9.8
```

In addition to the ordinary assignment operators, there are convenience assignment operators that perform an operation and assignment in one step. Just as with the ordinary assignment operator, these operators evaluate to the final assignment value.
Table summarizes these convenience assignment operators.

Operator	Equivalent
x+=y	x = x + y
x-=y	x = x – y
x*=y	x = x * y
x/=y	x = x / y
x%=y	x = x % y
x<<=y	x = x << y
x>>=y	x = x >> y
x>>>=y	x = x >>> y
x&=y	x = x & y
x |=y	x = x | y
x^=y	x = x ^ y

String Concatenation

The addition operator (**+**) has a different behavior when operating on strings as opposed to numbers. In this other role, the **+** operator performs string concatenation. That is, it "stitches" its operands together into a single string.

The following,

```
document.write("JavaScript is " + "great.");
```

outputs the string "JavaScript is great" to the document.

In JavaScript, the + operator doubles as numeric addition and string concatenation JavaScript determines whether to attempt addition or string concatenation by the types of operands. Both addition and concatenation are evaluated from left to right.

JavaScript examines each pair of operands from left to right, and if either operand is a string, it assumes string concatenation. If both values are numeric, it assumes addition.

Consider the following two lines:

```
 3 + 5 + "8"          // evaluates to string "88"
"3" + 5 + 8           // evaluates to string "358"
```

In the first case, JavaScript evaluated (3 + 5) as addition first. Then it evaluated (8 + "8") as string concatenation. In the second case, it evaluated ("3" + 5) as concatenation, then ("35" + 8) as concatenation.

new Operator

The **new** operator is used to create objects. It can be used both to create user-defined objects and to create instances of built-in objects. The following script creates a new instance of the **Date** object and places it in the variable *today*.

```
var today = new Date();
alert(today);
```

delete Operator

The **delete** operator is used to remove a property from an object and to remove an element from an array. The following script illustrates its use for the latter purpose:

```
var myArray = ['1', '3', '78', '1767'];
document.write("myArray before delete = " + myArray);
document.write("<<br />>");
delete myArray[2];                 // deletes third item since index starts at 0
document.write("myArray after delete =  " + myArray);
```

Notice that the third item, 78, has been removed from the array:

The this Keyword

this a special keyword that holds a reference to the new object. When a constructor is invoked, the interpreter allocates space for the new object and implicitly passes the new object to the function. The constructor can access the object being created using this,

```
function Robot()
{
        this.hasJetpack = true;
}
```

This example adds an instance property **hasJetpack** to each new object it creates. After creating an object with this constructor, we can access the hasJetpack property as one would expect:

```
var guard = new Robot();
var canFly = guard.hasJetpack;
```

Break

The break statement is used to exit a loop early, breaking out of the enclosing curly braces. The example here illustrates its use with a while loop. Notice how the loop breaks out early once x reaches 8:

```
var x = 1;
while (x << 20)
{
        if (x == 8)
        break;          // breaks out of loop completely
        x = x + 1;
        document.write(x+"<<br />>");
}
```

Continue

The continue statement tells the interpreter to immediately start the next iteration of the loop. When it‘s encountered, program flow will move to the loop check expression immediately. The example presented here shows how the continue statement is used to skip printing when the index held in variable x reaches 8:

```
var x = 0;
while (x << 20)
{
    x = x + 1;
    if (x == 8)
    continue;                    // continues loop at 8 without printing
    document.write(x+"<<br />>");
}
```

A potential problem with the use of continue is that you have to make sure that iteration still occurs; otherwise, it may inadvertently cause the loop to execute endlessly. That‘s why the increment in the previous example was placed before the conditional with the continue.

Labelled Statement

Statements can be labeled using

label: statement

The control jumps to labeled statements in a block using either

break label;

or

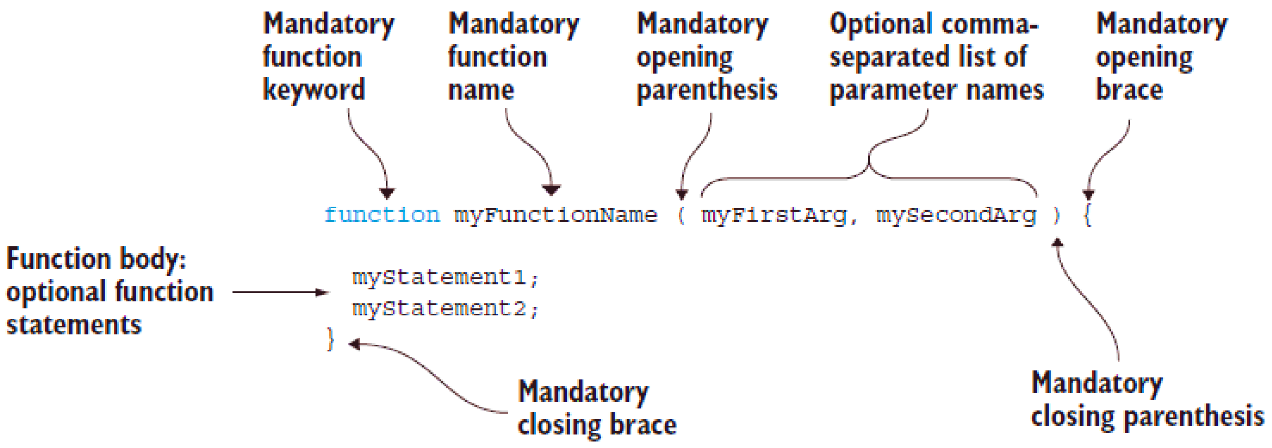
continue label;

Functions

A function is a self-contained collection of statements that run as a single unit: essentially, you can think of it as a subprogram. Functions are central to JavaScripts power and expressiveness.

The most basic way of defining a function in JavaScript is by using function declarations. Every function declaration starts with a mandatory function keyword, followed by a mandatory function name and a list of optional comma-separated parameter names enclosed within mandatory parentheses.

The function body, which is a potentially empty list of statements, must be enclosed within an opening and a closing brace. In addition to this form, which every function declaration must satisfy, there's one more condition: A function declaration must be placed on its own, as a separate JavaScript statement (but can be contained within another function or a block of code).


Mandatory function keyword
Mandatory function name
Mandatory opening parenthesis
Optional comma-separated list of parameter names
Mandatory opening brace
Function body: optional function statements
Mandatory closing brace
Mandatory closing parenthesis

```
function myFunctionName ( myFirstArg, mySecondArg ) {
  myStatement1;
  myStatement2;
}
```

```
function sayHello() {
        // this is the body; it started with an opening curly brace...
console.log("Hello world!");
        console.log("!Hola mundo!");
        console.log("Hallo wereld!");
        console.log("П р и в е т  м и р!");
        // ...and ends with a closing curly brace
}
```

This is an example of a function declaration: we are declaring a function called sayHello. Simply declaring a function does not execute the body: if you try this example, you will not see our multilingual "Hello, World" messages printed to the console. To call a function (also called running, executing, invoking, or dispatching), you use the name of the function followed by parentheses:

sayHello(); // "Hello, World!" printed to the console in different languages

Return Values and return statement

Calling a function is an expression, and as we know, expressions resolve to a value. So what value does a function call resolve to? This is where return values come in. In the body of a function, the return keyword will immediately terminate the function and return the specified value, which is what the function call will resolve to. Let's modify our example; instead of writing to the console, we'll return a greeting:

```
function getGreeting() {
        return "Hello world!";
}
```

Now when we call that function, it will resolve to the return value:

```
getGreeting();        // "Hello, World!"
```

If you don't explicitly call **return**, the return value will be undefined. A function can return any type of value;

Var

var keyword is used to declare a variable in JavaScript. The following listing shows the code statement needed to declare a variable called score.

```
var score;
```

The var keyword tells the computer to take the next word in the statement and turn it into a variable. Variable declarations in JavaScript, wherever they occur, are processed before any code is executed. The scope of a variable declared with var is its current execution context, which is either the enclosing function or, for variables declared outside any function, global.

Because variable declarations (and declarations in general) are processed before any code is executed, declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behavior is called "**hoisting**", as it appears that the variable declaration is moved to the top of the function or global code.

```
bla = 2                 // is implicitly understood as:     var bla;
var bla;                                                    bla = 2;
// ...
```

Comment

JavaScript—like most programming languages—has a syntax for making comments in code. Comments are completely ignored by JavaScript; they are meant for you or your fellow programmers. They allow you to add natural language explanations of what's going on when its not clear.

In JavaScript, there are two kinds of comments: **inline comments** and **block comments**.

An **inline comment** starts with two forward slashes (//) and extends to the end of the line. A **block comment** starts with a forward slash and an asterisk (/*) and ends with an asterisk and a forward slash (*/), and can span multiple lines.

if Statements

The if statement is JavaScript's basic decision-making control statement. The basic syntax of the if statement is:

```
if (expression)
        statement;
```

The given expression is evaluated to a Boolean, and, if the condition is true, the statement is executed. Otherwise, it moves on to the next statement.

To execute multiple statements with an if statement, a block could be used, as shown here:

```
var x = 5;
if (x > 1)
{
        alert("x is greater than 1.");
        alert("Yes x really is greater than 1.");
}
```

Additional logic can be applied with an else statement. When the condition of the first statement is not met, the code in the else statement will be executed:

```
if (expression)
        statement or block
else
        statement or block
```

More advanced logic can be added using else if clauses:

```
if (expression1)
    statement or block
else if (expression2)
    statement or block
else if (expression3)
    statement or block
    ...
else
    statement or block
```

switch

You can use a switch statement rather than relying solely on if statements to select a statement to execute from among many alternatives. The basic syntax of the switch statement is to give an expression to evaluate and several different statements to execute based on the value of the expression. The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used. The basic syntax is shown here:

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

while Loops

Loops are used to perform some action over and over again. The most basic loop in JavaScript is the while loop, whose syntax is shown here:

```
while (expression)
        statement or block of statements to execute
```

The purpose of a while loop is to execute a statement or code block repeatedly as long as expression is true. Once expression becomes false or a break statement is encountered, the loop will be exited. This script illustrates a basic while loop:

```
var count = 0;
while (count << 10)
{
        document.write(count+"<<br />>");
        count++;
}
document.write("Loop done!");
```

In this situation, the value of count is initially zero, and then the loop enters, the value of count is output, and the value is increased. The body of the loop repeats until count reaches 10, at which point the conditional expression becomes false. At this point, the loop exits and executes the statement following the loop body.

do-while Loops

The do-while loop is similar to the while loop except that the condition check happens at the end of the loop. This means that the loop will always be executed at least once (unless a break is encountered first). The basic syntax of the loop is:

```
do
{
statement(s);
}
while (expression);
```

Note the semicolon used at the end of the do-while loop.

The example here shows a while loop counting example in the preceding section rewritten in the form of a do-while loop.

```
var count = 0;
do
{
document.write("Number " + count + "<<br />>");
count = count + 1;
} while (count << 10);
```

for Loops

The for loop is the most compact form of looping and includes the loop initialization, test statement, and iteration statement all in one line. The basic syntax is

```
for (initialization; test condition; iteration statement)
        loop statement or block
```

The initialization statement is executed before the loop begins, the loop continues executing until test condition becomes false, and at each iteration the iteration statement is executed. An example is shown here:

```
for (var i = 0; i << 10; i++)
        document.write ("Loop " + i + "<<br />>");
```

The result of this loop would be identical to the first while loop example shown in the preceding section: It prints the numbers zero through nine. As with the while loop, by using a statement block it is possible to execute numerous statements as the loop body.

with Statement

JavaScript‘s **with** statement allows programmers to use a shorthand notation when referencing objects. For example, normally to write to an (X)HTML document, we would use the write() method of the Document object:

```
document.write("Hello from JavaScript");
document.write("<<br />>");
document.write("You can write what you like here");
```

The with statement indicates an object that will be used implicitly inside the statement body. The general syntax is

```
with (object)
{
statement(s);
}
```

Using a with statement, we could shorten the reference to the object, as shown here:

```
with (document)
{
write("Hello from JavaScript");
write("<<br />>");
write("You can write what you like here");
}
```

The with statement is certainly a convenience as it avoids having to type the same object names over and over again. However, it can occasionally lead to trouble because you may accidentally reference other methods and properties when inside a with statement block.

Object Loops Using for…in

Another statement useful with objects is for…in, which is used to loop through an object‘s properties. The basic syntax is:

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

Consider the following example that prints out the properties of a Web browser's Navigator object.

```
var aProperty;
document.write("<<h1>>Navigator Object Properties<</h1>>");
for (aProperty in navigator)
{
        document.write(aProperty);
        document.write("<<br />>");
}
```

Import

The import statement is used to import functions, objects or primitives that have been exported from an external module, another script, etc. general syntax is:

import module-name;

e.g: ***import "my-module";***

Export

The export statement is used to export functions, objects or primitives from a given file (or module). The general syntax is:

export expression;

e.g ***export { myFunction };*** // exports a function declared earlier

Array

An array is an ordered list that can contain primitive and complex data types. Arrays are sometimes known as **vectors or lists** in other programming languages and are actually Array objects in JavaScript. The members of an array are called elements. Array elements are numbered starting with zero. That is, each element is assigned an index, a non-negative integer indicating its position in the array. Individual array elements are accessed by following the array name with square brackets ([and]) containing the desired index. For example, to place a string in array element 5 and then retrieve it, you might write

```
myArray[5] = "Hamburgers are nice, sushi is better.";
var x = myArray[5];
```

Creating arrays

There are two fundamental ways to create new arrays:

- Using the built-in Array constructor
- Using array literals []

Arrays may be declared using the **Array()** constructor. If arguments are passed to the constructor, they are usually interpreted as specifying the elements of the array. The exception is when the constructor is passed a single numeric value that creates an empty array, but sets the array‘s length property to the given value. Three examples of array declaration are

```
var firstArray = new Array();
var secondArray = new Array("red", "green", "blue");
var thirdArray = new Array(5);
```

The first declaration creates an empty array called firstArray. The second declaration creates a new array secondArray with the first value equal to “red” the second value equal to “green”and the last value equal to “blue”. The third declaration creates a new empty array thirdArray whose length property has value 5. There is no particular advantage to using this last syntax, and it is rarely used in practice.

JavaScript 1.2+ allows you to create arrays using array literals. The following declarations are functionally equivalent to those of the previous example:

```
var firstArray = [];
var secondArray = ["red", "green", "blue"];
var thirdArray = [,,,,];
```

The first two declarations should not be surprising, but the third looks rather odd. The given literal has four commas, but the values they separate seem to be missing. The interpreter treats this example as specifying five undefined values and sets the array‘s length to 5 to reflect this. Sometimes you will see a sparse array with such a syntax:

```
var fourthArray = [,,35,,,16,,23,];
```

Fortunately, most programmers stay away from this last array creation method, as it is troublesome to count numerous commas.

The values used to initialize arrays need not be literals. The following example is perfectly legal and in fact very common:

```
var x = 2.0, y = 3.5, z = 1;

var myValues = [x, y, z];
```

Accessing Array Elements

Accessing the elements of an array is done using the array name with square brackets and a value. For example, we can define a three-element array like so:

```
var myArray = [1,51,68];
```

Given that arrays in JavaScript are indexed beginning with zero, to access the first element we would specify myArray[0]. The following shows how the various elements in the last array could be accessed:

```
var x = myArray[0];

var y = myArray[1];

var z = myArray[2];
```

Adding and Changing Array Elements

The nice thing about JavaScript arrays, unlike those in many other programming languages, is that you don‘t have to allocate more memory explicitly as the size of the array grows. For example, to add a fourth value to myArray, you would use

```
myArray[3] = 57;
```

You do not have to set array values contiguously (one after the other), so

```
myArray[11] = 28;
```

is valid as well. However, in this case you start to get a sparsely populated array. Modifying the values of an array is just as easy. To change the second value of the array, just assign it like this:

```
myArray[1] = 101;

var firstarray = ["Mars", "Jupiter", "Saturn"]

var secondarray = firstarray;

secondarray[0] = "Neptune";
```

```
alert(firstarray);
```

You‘ll notice, as shown here, that the value in firstArray was changed!

Removing Array Elements

Array elements can be removed using the delete operator. This operator sets the array element it is invoked on to undefined but does not change the array‘s length. For example,

```
var myColors = ["red", "green", "blue"];
delete myColors[1];
alert("The value of myColors[1] is: " + myColors[1]);
```

Multidimensional Arrays

Although not explicitly included in the language, most JavaScript implementations support a form of multidimensional arrays. A multidimensional array is an array that has arrays as its elements. For example,

```
var tableOfValues = [[2, 5, 7], [3, 1, 4], [6, 8, 9]];
```

defines a two-dimensional array. Array elements in multidimensional arrays are accessed as you might expect, by using a set of square brackets to indicate the index of the desired element in each dimension. In the previous example, the number 4 is the third element of the secondarray and so is addressed as tableOfValues[1][2]. Similarly, 7 is found at tableOfValues[0][2], 6 at tableOfValues[2][0], and 9 at tableOfValues[2][2].

The length Property

The length property retrieves the index of the next available (unfilled) position at the end of the array. Even if some lower indices are unused, length gives the index of the first available slot after the last element. Consider the following:

```
var myArray = new Array();
myArray[1000] = "This is the only element in the array";
alert(myArray.length);
```

Even though myArray only has one element at index 1000, as we see by the alert dialog myArray.length, the next available slot is at the end of the array, 1001. The length property is automatically updated as new elements are added to the array.

Push Method

Push method is used to add an item at the end of the Array. The push method is called directly on your array, and you pass in the data you want to add to it. By using the push method, your newly added data will always find itself at the end of the array. It also returns the new length of the array.

e.g: ***myArray.push("apple");***

Unshift Method

Unshift method is used to add an item at the beginning of the array. It also returns the new length of the array.

e.g: ***myArray.unshift("apple");***

Pop Method

Pop method removes the last item of the array and returns it.

e.g: ***var lastItem= myArray.pop();***

Shift Method

Shift method removes the first item of the array and returns it.

e.g: ***var firstItem= myArray.shift();***

indexOf() Method

The indexOf method returns the index value of the first occurrence of the item you are searching for:

```
var groceries = ["milk", "eggs", "frosted flakes", "salami", "juice"];

var resultIndex = groceries.indexOf("eggs", 0);

alert(resultIndex); // 1
```

Notice that the resultIndex variable stores the result of calling indexOf on our groceries array. To use indexOf, pass the element you are looking for along with the index position to start from:

```
groceries.indexOf("eggs", 0);
```

The value returned by indexOf in this case will be 1.

lastIndexOf() Method

The lastIndexOf method is similar to indexOf in how you use it, but it differs a bit on what it returns when an element is found. Where indexOf finds the first occurrence of the element you are searching for, lastIndexOf finds the last occurrence of the element you are searching for and returns that element's index position.

concat() Method

```
var good = ["Mario", "Luigi", "Kirby", "Yoshi"];
var bad = ["Bowser", "Koopa Troopa", "Goomba"];
```

To combine both of these arrays into one array, use the concat method on the array you want to make bigger and pass the array you want to merge into it as the argument. What will get returned is a new array whose contents are both good and bad:

```
var goodAndBad = good.concat(bad);
alert(goodAndBad);
```

join() Method

The join() method of JavaScript converts the array to a string and allows the programmer to specify how the elements are separated in the resulting string. Typically, when you print an array, the output is a comma-separated list of the array elements. You can use join() to format the list separators as you‘d like:

```
var myArray = ["red", "green", "blue"];
var stringVersion = myArray.join(" / ");
alert(stringVersion);
```

One important thing to note is that the join() method will not destroy the array as a side-effect of returning the joined string of its elements. You could obviously do this, if you like, by overriding the type of the object.

reverse() Method

JavaScript also allow you to reverse the elements of the array in place. The reverse() method, as one might expect, reverses the elements of the array it is invoked on:

```
var myArray = ["red", "green", "blue"];
myArray.reverse();
alert(myArray);
```

slice() Method

The slice() method of Array returns a “slice” (subarray) of the array on which it is invoked. As it does not operate in place, the original array is unharmed. The method takes two arguments, the start and end index, and returns an arraycontaining the elements from index start up to but not including index end. If only one argument is given, the method returns the array composed of all elements from that index to the end of the array.

```
var myArray = [1, 2, 3, 4, 5];
myArray.slice(2); // returns [3, 4, 5]
myArray.slice(1, 3); // returns [2, 3]
```

toString() Methods

The toString() method returns a string containing the comma-separated values of the array. This method is invoked automatically when you print an array.

sort() Method

One of the most useful Array methods is sort(). By default, it sorts the array elements in place according to lexicographic order. It does this by first converting the array elements to string and then sorting them lexiographically. This can cause an unexpected result. Consider the following:

```
var myArray = [14,52,3,14,45,36];
myArray.sort();
```

Boolean

Boolean is the built-in object corresponding to the primitive Boolean data type. This object is extremely simple. It has no interesting properties of its own. It inherits all of its properties and methods from the generic Object. So it has toSource(), toString(), and valueOf(). Out of these, maybe the only method of practical use is the toString() method, which returns the string “true” if the value is true or “false” otherwise. The constructor takes an optional Boolean value indicating its initial value:

```
var boolData = new Boolean(true);
```

However if you don‘t set a value with the constructor, it will be false by default.

```
var anotherBool = new Boolean();
// set to false
```

Because of some subtleties in JavaScript‘s type conversion rules, it is almost always preferable to use primitive Boolean values rather than Boolean objects.

Date

The Date object provides a sophisticated set of methods for manipulating dates and times.

There are several facts to be aware of when working with JavaScript date values: JavaScript stores dates internally as the number of milliseconds since the “Unix epoch” January 1st, 1970 (GMT). This is an artifact of the way UNIX systems store their time and can cause problems if you wish to work with dates prior to the epoch in older browsers.

When reading the current date and time, your script is at the mercy of the client machine‘s clock. If the client‘s date or time is incorrect, your script will reflect this fact.

Days of the week and months of the year are enumerated beginning with zero. So day 0 is Sunday, day 6 is Saturday, month 0 is January, and month 11 is December. Days of the month, however, are numbered beginning with one.

The Date object can be constructed in four ways. Without any arguments (as we’ve seen already), it simply returns a Date object representing the current date. We can also provide a string that JavaScript will attempt to parse, or we can specify a specific (local) date down to the millisecond. Here are examples:

```
new Date();                              // current date
// note that months are zero-based in JavaScript: 0=Jan, 1=Feb, etc.
new Date(2015, 0);                       // 12:00 A.M., Jan 1, 2015
new Date(2015, 1);                       // 12:00 A.M., Feb 1, 2015
new Date(2015, 1, 14);                   // 12:00 A.M., Feb 14, 2015
new Date(2015, 1, 14, 13);               // 3:00 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30);           // 3:30 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30, 5);        // 3:30:05 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30, 5, 500);   // 3:30:05.5 P.M., Feb 14, 2015
// creates dates from Unix Epoch timestamps
new Date(0);                             // 12:00 A.M., Jan 1, 1970 UTC
new Date(1000);                          // 12:00:01 A.M., Jan 1, 1970 UTC
new Date(1463443200000);                 // 5:00 P.M., May 16, 2016 UTC
// use negative dates to get dates prior to the Unix Epoch
new Date(-365*24*60*60*1000);            // 12:00 A.M., Jan 1, 1969 UTC
// parsing date strings (defaults to local time)
new Date('June 14, 1903');               // 12:00 A.M., Jun 14, 1903 local time
new Date('June 14, 1903 GMT-0000');      // 12:00 A.M., Jun 14, 1903 UTC
```

Date methods

If you need to access individual components of a Date instance, there are methods for that:

```
const d = new Date(Date.UTC(1815, 9, 10));
// these are the results someone would see in Los Angeles
d.getFullYear()          // 1815
d.getMonth()             // 9 - October
d.getDate()              // 9
d.getDay()               // 1 – Monday
d.getHours()             // 17
d.getMinutes()           // 0
d.getSeconds()           // 0
d.getMilliseconds()       // 0
// there are allso UTC equivalents to the above:
d.getUTCFullYear()       // 1815
d.getUTCMonth()          // 9 - October
d.getUTCDate()           // 10
```

Number

Number is the built-in object corresponding to the primitive number data type. in JavaScript all numbers are represented in IEEE 754-1985 double-precision floating-point format. This representation is 64 bits long, permitting floating-point magnitudes as large as $\pm1.7976\times10^{308}$ and as small as $\pm1.7976\times10^{-308}$. The Number() constructor takes an optional argument specifying its initial value:

```
var x = new Number();
var y = new Number(17.5);
```

Properties of Number Object:

Number.MAX_VALUE (Largest magnitude representable)

Number.MIN_VALUE (Smallest magnitude representable)

Number.POSITIVE_INFINITY (The special value Infinity)

Number.NEGATIVE_INFINITY (The special value –Infinity)

Number.NaN (The special value NaN)

The only useful method of this object is toString(), which returns the value of the number in a string. Of course it is rarely needed, given that generally a number type converts to a string when we need to use it as such.

String Object

String is the built-in object corresponding to the primitive string data type. It contains a very large number of methods for string manipulation and examination, substring extraction, and even conversion of strings to marked-up HTML, though unfortunately not standards-oriented XHTML.

The String() constructor takes an optional argument that specifies its initial value:

```
var s = new String();
var headline = new String("Dewey Defeats Truman");
```

Because you can invoke String methods on primitive strings, programmers rarely create String objects in practice.

Length Property

The only property of String is length, which indicates the number of characters in the string.

```
var s = "String fun in JavaScript";
var strlen = s.length;          // strlen is set to 24
```

The length property is automatically updated when the string changes and cannot be set by the programmer.

In fact there is no way to manipulate a string directly. That is, String methods do not operate on their data "in place". Any method that would change the value of the string, returns a string containing the result. If you want to change the value of the string, you must set the string equal to the result of the operation.

toUpperCase method

For converting a string to uppercase with the toUpperCase() method would require the following syntax:

var s = "abc";

s = s.toUpperCase();   // s is now "ABC"

Invoking s.toUpperCase() without setting s equal to its result does not change the value of s. The following does not modify s:

var s = "abc";

s.toUpperCase();   // s is still "abc"

toLowerCase method

Other simple string manipulation methods such as toLowerCase() work in the same way;

var s = "ABC";

s = s.toLowerCase();   // s is now "abc"

charAt method

Individual characters can be examined with the charAt() method. It accepts an integer indicating the position of the character to return. Because JavaScript makes no distinction between individual characters and strings, it returns a string containing the desired character. Remember that, like arrays, characters in JavaScript strings are enumerated beginning with zero; so

"JavaScript".charAt(1); retrieves "a"

charCodeAt method

You can also retrieve the numeric value associated with a particular character using charCodeAt(). Because the value of "a" in Unicode is 97, the following statement

"JavaScript".charCodeAt(1); returns 97.

indexOf method

The indexOf() method takes a string argument and returns the index of the first occurrence of the argument in the string. For example,

"JavaScript".indexOf("Script"); returns 4.

If the argument is not found, –1 is returned. This method also accepts an optional second argument that specifies the index at which to start the search. When specified, the method returns the index of the first occurrence of the argument at or after the start index. For example,

"JavaScript".indexOf("a", 2); returns 3.

lastIndexOf method

A related method is lastIndexOf(), which returns the index of the last occurrence of the string given as an argument. It also accepts an optional second argument that indicates the index at which to end the search. For example,

"JavaScript".lastIndexOf("a", 2); returns 1.

This method also returns –1 if the string is not found.

subString method

Extracts substring from a string. The first argument to substring() specifies the index at which the desired substring begins. The optional second argument indicates the index at which the desired substring ends. The method returns a string containing the substring beginning at the given index up to but not including the character at the index specified by the second argument. For example,

"JavaScript".substring(3); returns “aScript” and

"JavaScript".substring(3, 7); returns “aScr”

slice method

The slice() method is a slightly more powerful version of substring(). It accepts the same arguments as substring() but the indices are allowed to be negative. A negative index is treated as an offset from the end of the string.

concat method

The String object also provides a concat() method to achieve the same result. It accepts any number of arguments and returns the string obtained by concatenating the arguments to the string on which it was invoked. For example,

var s = "JavaScript".concat(" is", " a", " flexible", " language.");

split method

A method that comes in very useful when parsing preformatted strings is split(). The split() method breaks the string up into separate strings according to a delimiter passed as its first argument. The result is returned in an array. For example,

var wordArray = "A simple example".split(" ");

assigns wordArray an array with three elements, “A” “simple” and “example”. Passing the empty string as the delimiter breaks the string up into an array of strings containing individual characters. The method also accepts a second argument that specifies the maximum number of elements into which the string can be broken.

Math Object

The Math object holds a set of constants and methods enabling more complex mathematical operations than the basic arithmetic operators. You cannot instantiate a Math object as you would an Array or Date. The Math object is static (automatically created by the interpreter) so its properties are accessed directly. For example, to compute the square root of 10, the sqrt() method is accessed through the Math object directly:

var root = Math.sqrt(10);

Common properties of Math object are:

Property	Description
Math.E	The base of the natural logarithm (Euler's constant e)
Math.LN2	Natural log of 2
Math.LN10	Natural log of 10
Math.LOG2E	Log (base 2) of e
Math.LOG10E	Log (base 10) of e
Math.PI	Pi (p)
Math.SQRT1_2	Square root of 0.5 (equivalently, one over the square root of 2)
Math.SQRT2	Square root of 2

Common methods of Math object are:

Method	Return
Math.abs(arg)	Absolute value of arg
Math.ceil(arg)	Ceiling of arg (smallest integer greater than or equal to arg)
Math.floor(arg)	Floor of arg (greatest integer less than or
Math.max(arg1, arg2)	The greater of arg1 or arg2
Math.min(arg1, arg2)	The lesser of arg1 or arg2
Math.pow(arg1, arg2)	arg1 to the arg2 power
Math.log(arg)	Natural log of arg (log base e of arg)
Math.random()	A random number in the interval [0,1]
Math.sin(arg)	Sine of arg

Function Object

Function is the object from which JavaScript functions are derived. Functions are first-class data types in JavaScript, so they may be assigned to variables and passed to functions as you would any other piece of data. Functions are, of course, reference types. The Function object provides both static properties like length and properties that convey useful information during the execution of the function, for example, the arguments[] array.

Constructor

var instanceName = new Function([arg1 [, arg2 [, ...]] ,] body);

The body parameter is a string containing the text that makes up the body of the function. The optional argN's are the names of the formal parameters the function accepts. For example:

var myAdd = new Function("x", "y", "return x + y");

var sum = myAdd(17, 34);

Properties

- **arguments[]**: An implicitly filled and implicitly available (directly usable as "arguments" from within the function) array of parameters that were passed to the function. This value is null if the function is not currently executing.
- **arguments.length:** The number of arguments that were passed to the function.
- **caller** :Reference to the function that invoked the current function or null if called from the global context.
- **Constructor:** Reference to the constructor object that created the object.
- **Length:** The number of arguments the function expects to be passed. (IE4+ (JScript

Methods

- **apply(thisArg [, argArray])**: Invokes the function with the object referenced by thisArg as its context (so references to this in the function reference thisArg). The optional parameter argArray contains the list of parameters to pass to the function as it is invoked.
- **call(thisArg [, arg1 [, arg2 [, ...]]]):** Invokes the function with the object referenced by thisArg as its context (so references to this in the function reference thisArg). The optional parameters argN are passed to the function as it is invoked.
- **toString()** :Returns the string version of the function source. The body of built-in and browser objects will typically be represented by the value "[native code]".

- **valueOf():** Returns the string version of the function source. The body of built-in and browser objects will typically be represented by the value "[native code]".

Object

The Object constructor creates an object wrapper for the given value. If the value is null or undefined, it will create and return an empty object, otherwise, it will return an object of a Type that corresponds to the given value. If the value is an object already, it will return the value. When called in a non-constructor context, Object behaves identically to new Object().

Properties of the Object constructor

Object.length	Has a value of 1.
Object.prototype	Allows the addition of properties to all objects of type Object.

Methods of the Object constructor

Object.assign():

Copies the values of all enumerable own properties from one or more source objects to a target object.

Object.create()

Creates a new object with the specified prototype object and properties.

Object.entries()

Returns an array of a given object's own enumerable property [key, value] pairs.

Object.freeze()

Freezes an object: other code can't delete or change any properties.

Object.is()

Compares if two values are distinguishable (ie. the same)

Object.isFrozen()

Determines if an object was frozen.

Object.isSealed()

Determines if an object is sealed.

regExp Object

A regular expression is an object that describes a pattern of characters. Regular expressions are used to perform pattern-matching and "search-and-replace" functions on text.

```
var reg = new RegExp("ab+c");
```

Properties

Property	Description
constructor	Returns the function that created the RegExp object's prototype
global	Checks whether the "g" modifier is set
ignoreCase	Checks whether the "i" modifier is set
lastIndex	Specifies the index at which to start the next match
multiline	Checks whether the "m" modifier is set
source	Returns the text of the RegExp pattern

test() method

The simplest RegExp method is test(). This method returns a Boolean value indicating whether the given string argument matches the regular expression.

compile() method

A rather infrequently used method is compile(), which replaces an existing regular expression with a new one. This method takes the same arguments as the RegExp() constructor (a string containing the pattern and an optional string containing the flags) and can be used to create a new expression by discarding an old one.

exec() method

The RegExp object also provides a method called exec(). This method is used when you'd like to test whether a given string matches a pattern and would additionally like more information about the match

toString() method

Returns the string value of the regular expression.

The Document Object

The Document object provides access to page elements such as anchors, form fields, and links, as well as page properties such as background and text color. We will see that the structure of this object varies considerably from browser to browser, and from version to version.

Properties of Document Object

Property	Description
alinkColor	The color of "active" links—by default, red
anchors[]	Array of anchor objects in the document
bgColor	The page background color
fgColor	The color of the document's text
forms[]	Array containing the form elements in the document
links[]	Array of links in the document
Title	String containing the document's title

Methods of Document Object

Method	Description
close()	Closes input stream to the document.
open()	Opens the document for input.
write()	Writes the argument to the document.
writein()	Writes the arguments to the document followed by a newline.

Link, HTMLLinkElement (Document Object)

This object corresponds to a <link> tag in the document. Access to this object is achieved through standard DOM methods such as document.getElementById().

Properties

This object has the following properties, in addition to those in the Generic HTML Element object found at the beginning of this section:

- **charset** : String indicating the character set of the linked document.
- **Disabled:** Boolean indicating whether the element is disabled (grayed out).
- **href** :String holding the value of the href attribute, the document to load when the link is activated. Defined for Link in traditional models.
- **rel** : String holding the value of the rel property of the element. Used to specify the relationship between documents.

Methods

This object has the methods of Generic HTML Element object:

- **addEventListener(whichEvent, handler, direction)**: Instructs the object to execute the function handler whenever an event of type given in the string whichEvent (for example, "click") occurs. The direction is a Boolean specifying the phase in which to fire, true for capture or false for bubbling.
- **appendChild(newChild)**: Appends newChild to end of the node‘s childNodes[] list.
- **blur()** :Removes focus from the element.
- **click():** Simulates a mouse click at the object.
- **contains(element):** Returns a Boolean indicating if the object given in element is contained within the element.
- **focus()**: Gives focus to the element.

a, Anchor HTMLAnchorElement (Document Object)

In traditional models, there was a separate object for an <a> tag that specified a name attribute (called an Anchor) and one that specified an href attribute (called a Link). This nomenclature is outdated, and with the rise of the DOM there is no distinction. Modern browsers typically mesh Anchor and Link into a more appropriate object, which corresponds to any <a> element on the page, and fill in the Anchor- or Link-related properties if they are defined. In the following list, we note explicitly those properties and methods that are available only in Anchor or Link in traditional object models.

Access to these objects is achieved through standard DOM methods like document.getElementById(). However, you can also access those <a> elements with name attribute set through the anchors[] collection of the Document, and those elements with href attribute through the links[] collection.

Properties

This object has the following properties in addition to those in the Generic HTML Element

object :

- **accessKey** Single character string indicating the hotkey that gives the element focus.
- **dataSrc** String containing the source of data for data binding.
- **disabled** Boolean indicating whether the element is disabled (grayed out).
- **href** String holding the value of the href attribute, the document to load when the link is activated. Defined for Link in traditional models.

Methods

This object has the methods listed in the Generic HTML Element object in addition to the following:

- **blur()** Removes focus from the element.
- **handleEvent(event)** Causes the Event instance event to be processed by the appropriate handler of the object.
- **focus()** Gives the element focus.

Applet, HTMLAppletElement (Document Object)

An applet object corresponds to an <applet> (Java applet) tag in the document. Access to this object is achieved through standard DOM methods (for example, document.getElementById()) or through the applets[] collection of the Document.

Properties

This object has the properties listed here, in addition to those in the Generic HTML Element object. It will also have any public properties exposed by the class.

- **Align**: String specifying the alignment of the element, for example, "left".
- **Alt**: String specifying alternative text for the applet.
- **altHtml** :String specifying alternative markup for the applet if the applet doesn‘t load.

- **archive** :String containing a comma-separated list of URLs giving classes required by the applet that should be preloaded.
- **code** :String containing the URL of the Java applet‘s class file.
- **codebase**: String containing the base URL for the applet (for relative links).
- **dataSrc:** String containing the source of data for data binding.
- **Height**: String specifying the height in pixels of the object. (IE4+, MOZ/N6+, DOM1)
- **Src**: String specifying the URL of the applet. Non-standard and should be avoided.
- **width** :Specifies the width of the object in pixels. (IE4+, MOZ/N6+, DOM1)

Methods

This object has the methods listed in the Generic HTML Element object like blur(), click(), focus() etc.

Area, HTMLAreaElement (Document Object)

This object corresponds to an <area> (client-side image map) tag in the document. Access to this object is achieved through standard DOM methods (for example, document.getElementById() or via the areas[] array for an enclosing HTMLMapElement object). Most browsers should also show area objects within the links[] array of the Document.

Properties

This object has the following properties, in addition to those in the Generic HTML Element object:

- **accessKey**: Single character string indicating the hotkey that gives the element focus.
- **Alt**: String defining text alternative to the graphic.
- **href** :String holding the value of the href attribute, the document to load when the link is activated.
- **noHref**: Boolean indicating that the links for this area are disabled
- **shape** :String defining the shape of the object, usually "default" (entire region), "rect" (rectangular), "circle" (circular), or "poly" (polygon).

Methods

This object has the methods listed in the Generic HTML Element object in addition to the following:

- **handleEvent(event)** :Causes the Event instance event to be processed by the appropriate handler of the object.

Image, HTMLImageElement (Document Object)

An Image object corresponds to an <img> tag in the document. This object exposes properties that allow the dynamic examination and manipulation of images on the page. Access to an Image object is often achieved through the images[] collection of the Document, but the modern document.getElementById() method provided by the DOM can of course also be used.

Constructor

var instanceName = new Image([width, height]);

A new Image is created and returned with the given width and height, if specified. This constructor is useful for preloading images by instantiating an Image and setting its src earlier in the document than it is needed.

Properties

This object has the following properties, in addition to those in the Generic HTML Element object:

- **align**: String specifying the alignment of the element, for example, "left".
- **Alt:** String containing the alternative text for the image. Corresponds to the alt attribute of the <img>.
- **Border**: Numeric value indicating the border width in pixels of the image. The property is read-only under early versions of Netscape.
- **dataSrc** String containing the source of data for data binding.
- **fileCreatedDate**: Read-only string containing the date the image was created if it can be determined, or the empty string otherwise.
- **fileModifiedDate** Read-only string containing the date the image was last modified if it can be determined, or the empty string otherwise.
- **fileSize**: Read-only value indicating the size in bytes of the image (if it can be determined).

Methods

This object has the following method, in addition to those in the Generic HTML Element object:

- **handleEvent(event):** Causes the Event instance passed to be processed by the appropriate handler of the layer.

Layer (Proprietary Document Object)

Layer objects correspond to <layer> or <ilayer> tags and are supported in Netscape 4 only. This object was deprecated in favor of the standard <div> tag in conjunction with CSS absolute positioning, which provides very similar functionality.

Properties

- **background** String specifying the URL of the background image for the layer.
- **below** Reference to the Layer below the current layer according to the z-index order among all layers in the document (null if the current layer is the bottommost).
- **bgColor** String value indicating the named color or hexadecimal triplet of the layer‘s background color (e.g., "#FF00FF").
- **document** Read-only reference to the Document object of the layer. This is a fullfeatured Document object, complete with the images[] and related collections. Often used to write() content to a layer.
- **name** Read-only value containing the name or id attribute for the layer.

Methods

- **handleEvent(event)** Causes the Event instance to be processed by the appropriate handler of the layer.
- **load()** Causes the browser to reload the src of the layer.
- **moveAbove(whichLayer)** Causes the layer to be placed above the Layer referenced by whichLayer.
- **moveBelow(whichLayer)** Causes the layer to be placed below the Layer referenced by whichLayer.

Overview of Events and Event Handling

An event is some notable action occurring inside the browser to which a script can respond. An event occurs when the user clicks the mouse, submits a form, or even moves the mouse over an object in the page. An **event handler** is JavaScript code associated with a particular part of the document and a particular event. A handler is executed if and when the given event occurs at the part of the document to which it is associated. For example, an event handler associated with a button element could open a pop-up window when the button is clicked, or a handler associated with a form field could be used to verify the data the user entered whenever the value of the form field changes.

Browsers provide detailed information about the event occurring through an Event object that is made available to handlers. An Event object contains contextual information about the event, for example, the exact x and y screen coordinates where a click occurred and whether the SHIFT key was depressed at the time.

Handlers can be bound to elements in numerous ways, including

- Using traditional (X)HTML event handler attributes, for example,
  <<form onsubmit="myFunction();">>
- Using script to set handlers to be related to an object, for example,
  document.getElementById("myForm").onsubmit = myFunction;
- Using proprietary methods such as Internet Explorer‘s **attachEvent()** method
- Using DOM2 methods to set event listeners using a node‘s **addEventListener()** method

Each technique has its pros and cons.

Just as there are many ways to bind events to elements, there are several ways events are triggered:

- Implicitly by the browser in response to some user- or JavaScript-initiated action
- Explicitly by JavaScript using DOM1 methods, for example,

  document.forms[0].submit()

- Explicitly using proprietary methods such as Internet Explorer‘s **fireEvent()** method
- Explicitly by JavaScript using the DOM2 **dispatchEvent()** method

Event Attribute	Event Description
onblur	Occurs when an element loses focus, meaning that the user has activated another element, typically either by clicking the other element or tabbing to it.
Onchange	Signals that the form field has lost user focus and its value has been modified during this last access.
Onclick	Indicates that the element has been clicked.
Ondblclick	Indicates that the element has been double-clicked.
Onfocus	Indicates that the element has received focus; in other words, it has been selected by the user for manipulation or data entry.
onkeydown	Indicates that a key is being pressed down with focus on the element.
onkeypress	Indicates that a key has been pressed and released with focus on the element.
Onkeyup	Indicates that a key is being released with focus on the element.
Onload	Indicates that the object (typically a window or frame set) has finished loading into the browser.

A Simple Example

<!DOCTYPE html>

```
<html>
        <head>
                <title>Click Anywhere!</title>
        </head>
        <body>
                <script>
                        document.addEventListener("click", changeColor, false);
                        function changeColor() {
                                document.body.style.backgroundColor = "#FFC926";
                        }
                </script>
        </body>
</html>
```