1. Introduction to Scripting – Client Side Scripting vs. Server Side Scripting

What is Scripting?

A **scripting language** is a command set for controlling some specific piece of hardware, software, or operating system, often with rudimentary and in some cases more advanced programming-like control flow constructs. Stored sets of commands, especially those stored in files, can be generically called **scripts**. So here is from where the word script came about.

Two kinds of scripting are there.

Client Side Scripting: the scripts that run on client machine to perform local operations without the intervention of the server.

Server Side Scripting: the scripts that run on Server machine in order to respond to the user requests.

1.1 Server Side Scripting

- A program running on a web server (server-side scripting) is used to generate the web content on various web pages, manage user sessions, and control workflow. Server responses may be determined by such conditions as data in a posted HTML form, parameters in the URL, the type of browser being used, the passage of time, or a database or server state.
- Such web pages are often created with the help of server-side languages such as ASP, ColdFusion, Go, JavaScript, Perl, PHP, Ruby, Python, WebDNA and other languages, by a Support server that can run on the same hardware as the web server. These server-side languages often use the Common Gateway Interface (CGI) to produce dynamic web pages. Two notable exceptions are ASP.NET, and JSP, which reuse CGI concepts in their APIs but actually dispatch all web requests into a shared virtual machine.
- Dynamic web pages are often cached when there are few or no changes expected and the page is anticipated to receive considerable amount of web traffic that would create slow load times for the server if it had to generate the pages on the fly for each request.

List the advantages of Server-side scripting

- User can create one template for the entire website
- The site can use a content management system which makes editing simpler.
- Generally quicker to load than client-side scripting
- User is able to include external files to save coding.
- Scripts are hidden from view so it is more secure. Users only see the HTML output.
- User does not need to download plugins like Java or Flash.

List the disadvantages of Server-side scripting

- Many scripts and content management systems tools require databases in order to store dynamic data.
- It requires the scripting software to be installed on the server.
- The nature of dynamic scripts creates new security concerns, in some cases making it easier for hackers to gain access to servers exploiting code flaws.

1.2 Client-side scripting

- Client-side scripting is changing interface behaviors within a specific web page in response to mouse or keyboard actions, or at specified timing events. In this case, the dynamic behavior occurs within the presentation. The client-side content is generated on the user's local computer system.

- Such web pages use presentation technology called rich interfaced pages. Client-side scripting languages like JavaScript or ActionScript, used for Dynamic HTML (DHTML) and Flash technologies respectively, are frequently used to orchestrate media types (sound, animations, changing text, etc.) of the presentation. Client-side scripting also allows the use of remote scripting, a technique by which the DHTML page requests additional information from a server, using a hidden frame, XMLHttpRequests, or a Web service.

- The first widespread use of JavaScript was in 1997, when the language was standardized as ECMAScript and implemented in Netscape 3.

- Example:The client-side content is generated on the client's computer. The web browser retrieves a page from the server, then processes the code embedded in the page (typically written in JavaScript) and displays the retrieved page's content to the user.

- The innerHTML property (or write command) can illustrate the client-side dynamic page generation: two distinct pages, A and B, can be regenerated (by an "event response dynamic") as document.innerHTML = A and document.innerHTML = B; or "on load dynamic" by document.write(A) and document.write(B).

List the advantages of Client-side scripting

- Allow for more interactivity by immediately responding to users' actions.
- Execute quickly because they do not require a trip to the server.
- May improve the usability of Web sites for users whose browsers support scripts.
- Can give developers more control over the look and behaviour of their Web widgets.
- Can be substituted with alternatives (for example, HTML) if users' browsers do not support scripts are reusable and obtainable from many free resources.

List the disadvantages of Client-side scripting

- Not all browsers support scripts, therefore, users might experience errors if no alternatives have been provided.
- Different browsers and browser versions support scripts differently, thus more quality assurance testing is required.
- More development time and effort might be required (if the scripts are not already available through other resources).
- Developers have more control over the look and behaviour of their Web widgets; however, usability problems can arise if a Web widget looks like a standard control but behaves differently or vice-versa.

1.3 Difference between Server Side Scripting and Client Side Scripting

Client Side Scripting	**Server Side Scripting**
1. It is used to do some kind of basic operations in the browser when the page is displayed to the user.	It is used to perform some operations by the server machine.
2. It is run at Client machine.	It is run at Server machine.
3. This kind of scripts is written using JavaScript or VBScript.	This kind of scripts is written using ASP or PHP languages.
4. This kind of scripts does not require server.	This kind of scripts does require client as well as server.

1.4 List two languages used in Client-side scripting and Server-side scripting

Client Side Scripting Language:

1. Java script
2. VB Script

Server Side Scripting Language:

1. ASP
2. PHP
3. JSP

2. Write a note on JavaScript.

- JavaScript is a scripting language, that is, a lightweight programming language that is **interpreted** by the browser engine when the web page is loaded.

- HTML provides facility of designing web pages and JavaScript provides client-side programming facility to the user.
- Today's web sites need to go much beyond HTML. There is a definite need to allow users, browsing through a web site, to actually interact with the web site. The web site must be intelligent enough to accept users input and dynamically structure web page content, tailor made, to a user's requirements.
- This may be as simple as ensuring a web page delivered to a user, having a background color that the user is comfortable with or as complex as delivering a web page with special textual formatting for a user with visual disabilities.
- Users, who browse through a web site today, prefer to choose to view what interests them. Hence even the content of a web page needs to be dynamic, based on what a user wishes to see.
- This requires a web site development environment that will allow the creation of interactive web pages. These web pages will accept input from a user. Based on the input received return customized web pages, both in content and presentation, to the user.
- In the absence of any user input the web site must be intelligent enough to return a default web page containing predetermined information and text formatting.
- This calls for a web site development environment with coding techniques capable of accepting a client's requests and processing these requests. The result of the processing being passed back to the client via standard HTML pages.
- The need to return standard HTML pages, that map to a user's input, is due to the fact that browsers use HTTP to communicate with a web server and are designed to interpret and render HTML on a client's machine.
- Capturing user requests is traditionally done via a Form. Hence the web site development environment needs to have the facilities to create forms.
- After a form captures user input, the form must have a *built in* technique for sending the information captured back to the web server for processing. Processing user requests is generally done via scripts *(small programs)* that are based on the server.
- The web site development should also provide the facility for Validating user input. Invalid user input, will either cause data to be sent back from the web server to the browser, which is not what the user wants or give rise to an error message being sent back to the browser from the web server. Neither of which would really attract repeat visits the web.

- Hence, the web site development environment must also facilitate coding which runs in a browser at client side for data validation. Most development environments offer standard constructs for data validation. Standard programming constructs are:
  - Condition checking constructs
  - Case checking constructs
  - Super controlled Loop constructs

3. Write a note on Applications of JavaScript

Server-less CGIs. I use this term to describe processes that, were it not for JavaScript, would be programmed as CGIs on the server, yielding slow performance because of the interactivity required between the program and user. This includes tasks such as small data collection lookup, modification of images, and generation of HTML in other frames and windows based on user input.

Data entry validation. If form fields need to be filled out for processing on the server, I let clientside scripts prequalify the data entered by the user.

Dynamic HTML interactivity. It's one thing to use DHTML's capabilities to position elements precisely on the page; you don't need scripting for that. But if you intend to make the content dance on the page, scripting makes that happen.

CGI prototyping. Sometimes you want a CGI program to be at the root of your application because it reduces the potential incompatibilities among browser brands and versions. It may be easier to create a prototype of the CGI in client-side JavaScript. Use this opportunity to polish the user interface before implementing the application as a CGI.

Offloading a busy server. If you have a highly trafficked web site, it may be beneficial to convert frequently used CGI processes to client-side JavaScript scripts. After a page is downloaded, the server is free to serve other visitors. Not only does this lighten server load, but users also experience quicker response to the application embedded in the page.

Adding life to otherwise-dead pages. HTML by itself is pretty flat. Adding a blinking chunk of text doesn't help much; animated GIF images more often distract from, rather than contribute to, the user experience at your site. But if you can dream up ways to add some interactive zip to your page, it may engage the user and encourage a recommendation to friends or repeat visits.

creating web pages that "think." If you let your imagination soar, you may develop new, intriguing ways to make your pages appear "smart."

4. **Discuss advantages of JavaScript in brief.**

An Interpreted Language: JavaScript is an interpreted language, which requires no compilation steps. This provides an easy development process. The syntax is completely interpreted by the browser just as it interprets HTML tags.

Embedded Within HTML: JavaScript does not require any special or separate editor for programs to be written, edited or compiled. It can be written in any text editor like Notepad, along with appropriate HTML tags, and saved as filename.htmL HTML files with embedded JavaScript commands can then be read and interpreted by any browser that is JavaScript enabled.

Minimal Syntax - Easy to Learn: By learning just a few commands and simple rules of syntax, complete applications can be built using JavaScript.

Quick Development: Because JavaScript does not require time-consuming compilations, scripts can be developed in a short period of time. This is enhanced by the fact that many GUI interface features, such as alerts, prompts, confirm boxes, and other GUI elements, are handled by client side JavaScript, the browser and HTML code.

Designed for Simple, Small Programs: It is well suited to implement simple, small programs (for example, a unit conversion calculator between miles and kilometers, or pounds and kilograms). Such programs can be easily written and executed at an acceptable speed using JavaScript. In addition, they can be easily integrated into a web page.

Performance: JavaScript can be written such that the HTML files are fairly compact and quite small. This minimizes storage requirements on the web server and download time for the client.
Additionally, because JavaScript programs are usually included in the same file as the HTML code for a web page, they require fewer separate network accesses.

Procedural Capabilities: Every programming language needs to support facilities such as Condition checking, Looping and Branching. JavaScript provides syntax, which can be used to add such procedural capabilities to web page (filename.html) coding.

Designed for Programming User Events: JavaScript supports Object/Event based programming. JavaScript recognizes when a form Button is pressed. This event can have suitable JavaScript code attached, which will execute when the Button Pressed event occurs.
JavaScript can be used to implement context sensitive help. Whenever an HTML form's Mouse cursor Moves Over a button or a link on the page a helpful and informative message can be displayed in the status bar at the bottom of the browser window.

Easy Debugging and Testing: Being an interpreted language, scripts in JavaScript are tested line by line, and the errors are also listed as they are encountered, i.e. an appropriate error message along with the line number is listed for every error that is encountered. It is thus easy to locate errors, make changes, and test again without the overhead and delay of compiling.

Platform Independence / Architecture Neutral: JavaScript is a programming language that is completely independent of the hardware on which it works. It is a language that is understood by any JavaScript enabled browser. Thus, JavaScript applications work on any machine that has an appropriate JavaScripl enabled browser installed. This machine can be anywhere on the network.

5. Write a note on <script> tag

- To assist the browser in recognizing lines of code in an HTML document as belonging to a script, you surround lines of script code with a <script>...</script> tag set. This is common usage in HTML, where start and end tags encapsulate content controlled by that tag, whether the tag set is for a form or a paragraph.

- Depending on the browser, the <script> tag has a variety of attributes you can set that govern the script. One attribute, type, advises the browser to treat the code within the tag as JavaScript. Some other browsers accept additional languages (such as Microsoft's VBScript in Windows versions of Internet Explorer). The following setting is one that all modern scriptable browsers accept:

```
<script language="javascript">
```

  - Be sure to include the ending tag for the script. Lines of JavaScript code go between the two tags:

```
<script language="javascript">
        one or more lines of JavaScript code here
</script>
```

- If you forget the closing script tag, the script may not run properly, and the

- HTML elsewhere in the page may look strange.

Tag positions (where can we place <Script>…</Script> tag?)

- Where do these tags go within a document? The answer is, anywhere they're needed in the document. Most of the time, it makes sense to include the tags nested within the <head>...</head> tag set; other times, it is essential that you drop the script into a very specific location in the <body>...</body> section.

- Following example shows the outline of what may be the most common position of a <script> tag set in a document: in the <head> tag section.

- **Scripts in the Head**

```
<html>
<head>
<title>A Document</title>
<script type="text/javascript">
//script statement(s) here
...
</script>
</head>
<body>
</body>
</html>
```

- On the other hand, if you need a script to run as the page loads so that the script generates content in the page, the script goes in the <body> portion of the document, as shown in the following code.

- **A Script in the Body**

```
<html>
<head>
<title>A Document</title>
</head>
<body>
<script type="text/javascript">
//script statement(s) here
...
</script>
</body>
</html>
```

It's also good to know that you can place an unlimited number of <script> tag sets in a document. For example, the code given below shows a script in both the Head and Body portions of a document. Perhaps this document needs the Body script to create some dynamic content as the page loads, but the document also contains a button that needs a script to run later. That script is stored in the Head portion.

- **Scripts in the Head and Body**

```
<html>
<head>
<title>A Document</title>
<script language="javascript" >        //script statement(s) here
...
</script>
</head>
<body>
<script language="javascript">        //script statement(s) here
...
</script>
</body>
</html>
```

- You are not limited to one <script> tag set in either the Head or Body. You can include as many <script> tag sets in a document as are needed to complete your application. In following code, for example, two <script> tag sets are located in the Head portion. One set is used to load an external .js library; the other includes code specifically tailored to the current page.

- **Two Scripts in the Body**

```
<html>
<head><title>A Document</title></head>
<script language="javascript" src="js/jslibrary.js"></script>
<script language="javascript">
//script statement(s) here
...
</script>
<body>
</body>
</html>
```

1. DATATYPES

First of all let’s understand what is the meaning of Data? We might sometimes deal with some meaningful information. This meaningful information is called **Data**. Now Data Type means what is the type of data we are dealing with or working with. It may be numerical or textual.
JavaScript supports four primitive data types and it also supports complex types such as arrays and objects. Primitive types are types that can be assigned a single literal value such as number, string or Boolean value.
Literals are fixed values, which literally provide a value in a program.

Primitive data types that JavaScript supports are:

I. Number
II. String
III. Boolean
IV. Null

Let’s see each one in detail.

Number

This data type contains integer and floating point numbers.
Integer literals can be represented in JavaScript in decimal, hexadecimal, and octal form.
e.g. 33, 14, 67, 0x4d etc.
Floating-point literals are used to represent numbers that require the use of a decimal point, or very large or very small numbers that must be written using exponential notation. A floating-point number must consist of either a number containing a decimal point or an integer followed by an exponent.
e.g. 12.10, -35.8, 2E3

String

Consists of string values that are enclosed in single or double quotes.
JavaScript provides built-in support for strings. A string is a sequence of zero or more characters that are enclosed by double (“) or single(‘) quotes. If a string begins with a double quote it must end with a double quote. If a string begins with a single quote it must end with a single quote.
e.g. “Rahul”, ’34, Shyam Appts. , V.V. Nagar, Gujarat’

Boolean

Consists of the logical value *true* and *false*.
JavaScript supports a pure Boolean type that consists of the two values true and false. Logical operators can be used in Boolean expressions.
JavaScript automatically converts the Boolean values true and false into 1 and 0 when they are ued in numerical expressions.

Null

Consists of a single value, null, which identifies a null, empty or nonexistent reference.
The null value is common to all JavaScript types. It is used to set a variable to an initial value that is different from other valid values. Use of the null value prevents the sort of errors that result from using un-initialized variables. The null value is automatically converted to default values of other types when used in expression.

2. LITERALS

2.1 JavaScript: Integers literals

Description

An **integer** must have at least one digit (0-9).

- No comma or blanks are allowed within an integer.
- It does not contain any fractional part.
- It can be either positive or negative if no sign precedes it is assumed to be positive.

In JavaScript, integers can be expressed in three different bases.

1. Decimal (base 10)

Decimal numbers can be made with the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and there will be no leading zeros.
Example: 123, -20, 12345

2. Hexadecimal (base 16)

Hexadecimal numbers can be made with the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 and letters A, B, C, D, E, F or a, b, c, d, e, f. A leading 0x or 0X indicates the number is hexadecimal.
Example: 7b, -14, 3039

3. Octal (base 8)

Octal numbers can be made with the digits 0, 1, 2, 3, 4, 5, 6, 7. A leading 0 indicates the number is octal.
Example: 173, -24, 30071

2.2 JavaScript: Floating number literals

Description

A floating number has the following parts.

- A decimal integer.
- A decimal point ('.').
- A fraction.
- An exponent.

The exponent part is an "e" or "E" followed by an integer, which can be signed (preceded by "+" or "-").

Example of some floating numbers :

- 8.2935
- -14.72
- 12.4e3 [Equivalent to 12.4 x 10^{3}]
- 4E-3 [Equivalent to 4 x 10^{-3} => .004]

2.3 JavaScript: Boolean literals

The Boolean type has two literal values:

- true
- false

2.4 JavaScript: Object literals

Description

An object literal is zero or more pairs of comma-separated list of property names and associated values, enclosed by a pair of curly braces.

In JavaScript an object literal is declared as follows:

1. An object literal without properties:

```
var userObject = {}
```

2. An object literal with a few properties :

```
var student = {
First-name : "Suresy",
Last-name : "Rayy",
Roll-No : 12
};
```

Syntax Rules

Object literals maintain the following syntax rules:

- There is a colon (:) between property name and value.
- A comma separates each property name/value from the next.
- There will be no comma after the last property name/value pair.

2.5 JavaScript: String literals

Description

JavaScript has its own way to deal with string literals. A string literal is zero or more characters, either enclosed in single quotation (') marks or double **quotation** (") marks. You can also use + operator to join strings. The following are the examples of string literals :

- string1 = "w3resource.com"
- string1 = 'w3resource.com'
- string1 = "1000"

- string1 = "google" + ".com"

In addition to ordinary characters, you can include special characters in strings, as shown in the following table.
string1 = "First line. \n Second line."

3. VARIABLES

We can store data in two ways either permanently or temporarily. Permanently means we can store data onto a DVD or CD. Whereas temporarily means to store the data till our HTML page exists or till our program execution continues. So we need something to store the data temporarily in computer's memory. This is what we call it as Variables.

Variables are held in computer's memory. So it would be much, much easier to store and manipulate (retrieve) data.
We should keep some **rules** in our mind while creating variables (for variable names). Some of those rules are mentioned here.
We can't use certain names and characters for your variable names. Names we can't use are called *reserved* words. Reserved words are words that JavaScript keeps for its own use (for example, the word **var** or the word with). Certain characters are also forbidden in variable names: for example, the ampersand (&) and the percent sign (%). You are allowed to use numbers in your variable names, but the names must not begin with numbers. So 101myVariable is not okay, but myVariable101 . Let's look at some more examples. Invalid names include:

- with
- 99variables
- my%Variable
- theGood&theBad

Valid names include

- myVariable99
- myPercent_Variable
- the_Good_and_the_Bad

Declaration of variable is too much easy in JavaScript. By using **var** keyword we can declare like the example given below:

```
var myFirstVariable;
```

Initializing and assigning a value to a variable is also easy.

```
myFirstVariable=25;            or
myFirstVariable=”Hello!”;
```

Here in the first example variable myFirstVariable will contain the integer value, whereas in the second example the variable will contain the String value. So here type of data of variable clearly depends on the value which we are assigning to the variable.

Scope of Variable

What is meant by *scope*? Well, put simply, it's the scope or extent of a variable's availability — which parts of your code can access a variable and the data it contains. Any variables declared in a web page outside of a function will be available to all script on the page, whether that script is inside a function or otherwise — we term this a *global* or *page-level scope*.

However, variables declared inside a function are visible *only* inside that function — no code outside the function can access them. So, for example, you could declare a variable degCent in every function you have on a page *and* once on the page outside any function. However, you can't declare the variable *more* than once inside any one function or *more* than once on the page outside the functions. Note that reusing a variable name throughout a page in this way, although not illegal, is not standard good practice — it can make the code very confusing to read.

Function parameters are similar to variables: They can't be seen outside the function, and although you can declare a variable in a function with the same name as one of its parameters, it would cause a lot of confusion and might easily lead to subtle bugs being overlooked. It's therefore bad coding practice and best avoided, if only for the sake of your sanity when it comes to debugging! So what happens when the code inside a function ends and execution returns to the point at which the code was called? Do the variables defined within the function retain their value when you call the function the next time?

The answer is no: Variables not only have the scope property — where they are visible — but they also have a *lifetime*. When the function finishes executing, the variables in that function die and their values are lost, unless you return one of them to the calling code. Every so often JavaScript performs garbage collection, whereby it scans through the code and sees if any variables are no longer in use; if so, the data they hold are freed from memory to make way for the data of other variables.

Given that global variables can be used anywhere, why not make all of them global? Global variables are great when you need to keep track of data on a global basis. However, because they are available for modification anywhere in your code, it does mean that if they are changed incorrectly due to a bug, that bug could be anywhere within the code, making debugging difficult. It's best, therefore, to keep global variable use to a minimum, though sometimes they are a necessary evil — for example, when you need to share data among different functions.

4. TYPE CASTING

4.1 Converting Numbers to Strings

The global method String() can convert numbers to strings.

It can be used on any type of numbers, literals, variables, or expressions:

Example

```
String(x)        // returns a string from a number variable x
String(123)      // returns a string from a number literal 123
String(100 + 23)  // returns a string from a number from an expression
```

The Number method toString() does the same.

Example

```
x.toString()
(123).toString()
(100 + 23).toString()
```

4.2 Converting Booleans to Strings

The global method String() can convert booleans to strings.

```
String(false)     // returns "false"
String(true)      // returns "true"
```

The Boolean method toString() does the same.

```
false.toString()   // returns "false"
true.toString()    // returns "true"
```

4.3 Converting Dates to Strings

The global method String() can convert dates to strings.

```
String(Date())  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"
```

The Date method toString() does the same.

Example

```
Date().toString()  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"
```

4.4 Converting Strings to Numbers

The global method Number() can convert strings to numbers.

Strings containing numbers (like "3.14") convert to numbers (like 3.14).

Empty strings convert to 0.

Anything else converts to NaN (Not a Number).

```
Number("3.14")    // returns 3.14
Number(" ")       // returns 0
Number("")        // returns 0
Number("99 88")   // returns NaN
```

4.5 The Unary + Operator

The **unary + operator** can be used to convert a variable to a number:

Example

```
var y = "5";      // y is a string
var x = + y;      // x is a number
```

If the variable cannot be converted, it will still become a number, but with the value NaN (Not a Number):

Example

```
var y = "John";   // y is a string
var x = + y;      // x is a number (NaN)
```

4.6 Converting Booleans to Numbers

The global method Number() can also convert booleans to numbers.

```
Number(false)     // returns 0
Number(true)      // returns 1
```

4.7 Converting Dates to Numbers

The global method Number() can be used to convert dates to numbers.

```
d = new Date();
Number(d)          // returns 1404568027739
```

The date method getTime() does the same.

```
d = new Date();
d.getTime()        // returns 1404568027739
```

4.8 Automatic Type Conversion

When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type.

The result is not always what you expect:

```
5 + null    // returns 5         because null is converted to 0
"5" + null  // returns "5null"   because null is converted to "null"
"5" + 2     // returns "52"      because 2 is converted to "2"
"5" - 2     // returns 3         because "5" is converted to 5
"5" * "2"   // returns 10        because "5" and "2" are converted to 5 and 2
```

4.9 Automatic String Conversion

JavaScript automatically calls the variable's toString() function when you try to "output" an object or a variable:

```
document.getElementById("demo").innerHTML = myVar;
```

```
// if myVar = {name:"Fjohn"}  // toString converts to "[object Object]"
// if myVar = [1,2,3,4]       // toString converts to "1,2,3,4"
// if myVar = new Date()      // toString converts to "Fri Jul 18 2014 09:08:55 GMT+0200"
```

Numbers and booleans are also converted, but this is not very visible:

```
// if myVar = 123             // toString converts to "123"
// if myVar = true            // toString converts to "true"
// if myVar = false           // toString converts to "false"
```

5. OPERATORS

Arithmetic Operators

Arithmetic operators are the most familiar operators because they are used every day to solve common math problems.

In general, an operator requiring a single operand is referred to as a **Unary** operator and the operator that requires two operand is referred to as a **Binary** operator.

In arithmetic operators binary operators are as given below:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
%	Modulus
/	Division

e.g.
```
x = y+2;
z=y/2;
```

Whereas the Unary operators are as given below and they can be called as increment/decrement operators.

Operator	Description
++	Return the value then increment
--	Return the value then decrement

e.g.
```
x++;
y--;
```

Logical Operators

Logical operators are used to perform Boolean operations AND, OR, NOT. The logical operators are:

Operator	Description
&&	Logical AND
||	Logical OR
!	Logical NOT

Comparison Operators

Comparison operators are used to compare two values. The comparison operators are:

Operator	Description
==	Equal
!=	Not equal
<	Less then
>	Greater then
<=	Lesser or equal
>=	Greater or equal

Assignment Operators

The assignment operator is used to update the value of a variable. Some assignment operators are combined with other operators to perform a computation on the valule contained in a variable and then update the variable with the new value.

Operator	Description
=	Sets the variable on the left of the = operator to the value of the expression on its right.
+=	Increments the variable on the left hand of the += operator by the value of the expression on its right. When used with strings, the value to the right of the += operator is appended to the value of the variable on the left of the += operator.
	Decrements the variable on the left of the -= operator by the value of the expression on its right.
*=	Multiplies the variable on the left of the *= operator by the value of the expression on its left.
/=	Divides the variable on the left of the /= operator by the value of the expression on its left.
%=	Takes the modulus of the variable on the left of the %= operator by the value of the expression on its left.

Immediate if/ Conditional/ Ternary Operator

JavaScript supports the conditional operator. It is ? and : . The conditional expression operator is a ternary operator since it takes three operand, a condition to be evaluated and two alternative values to be returned on the truth or falsity of the condition.

Syntax:

Condition ***?*** *expr1* ***:*** *expr2*

Here if the condition is true the expr1 expression is to be executed and otherwise the expr2 expression is to be executed.

String Concatenation (+) Operator

String concatenation operator is the one which is used to concatenate (join) two strings.

E.g. “Hello” + “How are you?”

This expression will return a single string “HelloHow are you?”

new Operator

The new operator is used to create an instance of an object type.

E.g. myArray = new Array( );

6. DIALOG BOXES

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

Here we will see each dialog box one by one:

Alert Dialog Box:

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

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

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

Confirmation Dialog Box:

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

If the user clicks on OK button the window method confirm() will return true. If the user clicks on the Cancel button confirm() returns false. You can use confirmation dialog box as follows:

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

Prompt Dialog Box:
The prompt dialog box is very useful when you want to pop-up a text box to get user input. Thus it enable you to interact with the user. The user needs to fill in the field and then click OK.
This dialog box is displayed using a method called prompt() which takes two parameters (i) A label which you want to display in the text box (ii) A default string to display in the text box.
This dialog box with two buttons: OK and Cancel. If the user clicks on OK button the window method prompt() will return entered value from the text box. If the user clicks on the Cancel button the window method prompt() returns null.
You can use prompt dialog box as follows:

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

7. BUILT-IN FUNCTIONS

parseInt( )

parseInt( ) is a function provided by JavaScript. This function takes string as an argument and converts the string into an integer value.

E.g.
```
myVar = parseInt("123");
myVar = parseInt("58.4 abc");
```

Here in the first example the function parseInt( ) will convert the string "123" into a numerical integer value 123 and this value will be assigned to variable myVar . Same way in the second example also the string "58.4 abc" will be converted to the integer value 58 and this value will be assigned to the variable myVar.

parseFloat( )

Similar to the parseInt( ) function, the parseFloat( ) function will convert the string value to the float value. It means it takes a string as an argument and will convert it to the Float value.

Example:
```
var myVar = "56.8 degree centigrade"
var var2;
var2=parseFloat(myVar);
```

Here in this example the variable myVar contains the string, but as we used parseFloat( ) function and as an argument we have passed the variable myVar, the string value of variable myVar will be converted to the float value. Finally variable var2 will be assigned with a value 56.8 ok!.

isNaN( )

It is the function to check whether the argument provided to this function is not a number value or not.(not a number means the value other than number. It may be string or other value.) It returns the value true if the argument provided to this function is not a number, but in the case if it is a number the function will return false value.

Example:

i. myVar = isNaN("hello");

ii. myVar = isNaN("34");

In the first example the function will return true value because the string provided is not a number. And in the second example the return value will be false because we provided a number value to the function.

The Number() Method

Number() can be used to convert JavaScript variables to numbers:

Example

```
Number(true);         // returns 1
Number(false);        // returns 0
Number("10");         // returns 10
Number("  10");       // returns 10
Number("10  ");       // returns 10
Number(" 10  ");      // returns 10
Number("10.33");      // returns 10.33
Number("10,33");      // returns NaN
Number("10 33");      // returns NaN
Number("John");       // returns NaN
```

8. FLOW CONTROLS

1. Decision making

If Condition:

It is used when we want some expression or a set of expressions is to be executed while some condition is true.

E.g.
```
if(flag==0)
{       myVar=25;     }
```

If – else

It is used when we want an expression or a set of expressions is to be executed while some condition is true and if false some other set of expressions is to be executed.

E.g.
```
If(flag==0)
        myVar=25;
else
        myVar=45;
```

Else if

E.g.
```
If (flag==0)
        myVar=25;
else if(flag==1)
        myVar=35;
else if(flag==2)
        myVar=45;
else
        myVar = 95;
```

Nested if

e.g.
```
if( flag == 0)
{       if(var2==35)
                if(var3>=30)
                        myVar=40;
}
else
{       if(var2==35)
                if(var3>=30)
                        myVar=50;
}
```

Switch

It is used to select any one from a number of alternatives.

Syntax:

```
switch(eval)
{
    case 1:
          //FIRST ALTERNATIVE
          break;
     case 2:
           //SECOND ALTERNATIVE
           break;

     default
         //DEFAULT ACTION
}
```

2. <u>**Looping**</u>

For loop

The for loop is the most basic type of loop which you might have used in other basic programming languages like ANSI C.

Syntax:

```
for( expr1; condition; expr2)
{
//Javascript commands
}
```

Here the expr1 sets up a counter variable and assign the initial value. The declaration of the counter variable can also be done here. Condition specifies the final value for the loop to fire. expr2 specifies how the initial value of the counter variable is incremented or decremented.
e.g.

```
for( var num = 10; num >=1; num--)
{
//Javascript command
document.write(num);
}
```

While loop

The while loop provided the similar functionality as the for loop provides. The basic syntax for while loop is
Syntax

```
        while(condition)
        {
        //JavaScript commands
        }
```

Here condition is a valid JavaScript expression that evaluates to a Boolean value. It goes as long as the condition is true.

The Do/While Loop

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

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

Example

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

Example

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

1. ARRAY

- An array is similar to a normal variable, in that you can use it to hold any type of data. As you have already seen, a normal variable can only hold one piece of data at a time. For example, you can set myVariable to be equal to 25 like so:
    - myVariable = 25;
- and then go and set it to something else, say 35:
    - myVariable = 35;
- However, when you set the variable to 35, the first value of 25 is lost. The variable myVariable now holds just the number 35.
- The difference between such a normal variable and an array is that an array can hold *more than one* item of data at the same time. For example, you could use an array with the name myArray to store both the numbers 25 and 35. Each place where a piece of data can be stored in an array is called an *element*. How do you distinguish between these two pieces of data in an array? You give each piece of data an *index* value. To refer to that piece of data you enclose its index value in square brackets after the name of the array. For example, in an array called myArray, we can store both the values 25 and 35.
    - myArray[0] = 25;
    - myArray[1] = 35;
- Notice that the index values start at 0 and not 1. Why is this? Surely 1 makes more sense after all, we humans tend to say the first item of data, followed by the second item, and so on.
- Ok fine. Now the question is how to define an Array?
- We can define the array just by using new operator and Array( ) function of JavaScript. If we want to create myArray say of length 5.
    - var myArray;
    - myArray = new Array(5);
- and by default if we don't specify the size of the array then it would create the array of zero size. Remember in JavaScript the size of the array can be extended dynamically as we go on assigning next elements.
- In JavaScript it is not strict that an array can have a sequence of values of only same type, but in JavaScript we may assign any type of data as we want.
- E.g. myArray[0]=5;
    - myArray[1]="Good Morning";

Dense Arrays.

- A dense array is an array that has been created with each of its elements being assigned a specific value. Dense arrays are used exactly in the same manner as other arrays. They are declared and initialized at the same time.

- Listing the values of the array elements in the array declaration creates dense arrays. For example,
    - arrayName = new Array(value0, value1, ..., value**n**)
- In this array the elements are stored from values 0 to n. Hence the length of this array is n+1.
- As we already have seen that we can initialize array elements by using the index of it.
    - myArray[0]=5;
    - myArray[1]="Bharat";
    - myArray[2]= myArray[0] + myArray[1];
    - myArray[3] = "Hello";

2. USER-DEFINED FUNCTION

- JavaScript Function Syntax
- A function is written as a code block (inside curly { } braces), preceded by the **function** keyword:

```
function functionname()
{
some code to be executed
}
```

- The code inside the function will be executed when "someone" calls the function.
- The function can be called directly when an event occurs (like when a user clicks a button), and it can be called from "anywhere" by JavaScript code.

> JavaScript is case sensitive. The function keyword must be written in lowercase letters, and the function must be called with the same capitals as used in the function name.

Calling a Function with Arguments

- When you call a function, you can pass along some values to it, these values are called *arguments* or *parameters*.
- These arguments can be used inside the function.
- You can send as many arguments as you like, separated by commas (,)

```
myFunction(argument1,argument2)
```

- Declare the argument, as variables, when you declare the function:

```
function                                                    myFunction(var1,var2)
{
some                                                                         code
}
```

- The variables and the arguments must be in the expected order. The first variable is given the value of the first passed argument etc.
- Example

```
<button onclick="myFunction('Harry Potter','Wizard')">Try it</button>

<script>
function myFunction(name,job)
{
alert("Welcome " + name + ", the " + job);
}
</script>
```

- The function above will alert "Welcome Harry Potter, the Wizard" when the button is clicked.
- The function is flexible, you can call the function using different arguments, and different welcome messages will be given:
- Example

```
<button onclick="myFunction('Harry Potter','Wizard')">Try it</button>
<button onclick="myFunction('Bob','Builder')">Try it</button>
```

The example above will alert "Welcome Harry Potter, the Wizard" or "Welcome Bob, the Builder" depending on which button is clicked.

Functions With a Return Value

- Sometimes you want your function to return a value back to where the call was made.
- This is possible by using the *return* statement.
- When using the *return* statement, the function will stop executing, and return the specified value.
- Syntax

```
function myFunction()
{
var x=5;
return x;
}
```

- The function above will return the value 5.
- **Note:** It is not the entire JavaScript that will stop executing, only the function. JavaScript will continue executing code, where the function-call was made from.
- The function-call will be replaced with the return value:

```
var myVar=myFunction();
```

- The variable myVar holds the value 5, which is what the function "myFunction()" returns.
- You can also use the return value without storing it as a variable:

```
document.getElementById("demo").innerHTML=myFunction();
```

- The innerHTML of the "demo" element will be 5, which is what the function "myFunction()" returns.
- You can make a return value based on arguments passed into the function:
- Example

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

```
function                                                myFunction(a,b)
{
return                                                             a*b;
}
document.getElementById("demo").innerHTML=myFunction(4,3);
```

- The innerHTML of the "demo" element will be:

12

- The return statement is also used when you simply want to exit a function. The return *value* is optional:

```
function                                                myFunction(a,b)
{
if                                                                (a>b)
                                                                      {
 return;
                                                                      }
x=a+b
}
```

- The function above will exit the function if a>b, and will not calculate the sum of a and b.

<u>Local JavaScript Variables</u>

- A variable declared (using var) within a JavaScript function becomes **LOCAL** and can only be accessed from within that function. (the variable has local scope).
- You can have local variables with the same name in different functions, because local variables are only recognized by the function in which they are declared.
- Local variables are deleted as soon as the function is completed.

<u>Global JavaScript Variables</u>

- Variables declared outside a function, become **GLOBAL**, and all scripts and functions on the web page can access it.

3. STRING OBJECT

Using JavaScript Objects

- We can simply create the object in JavaScript by using the new keyword.

e.g.
```
myArray = new Array( );
myDate = new Date( );
myString = new String("Hello");
```

Using an Object's Properties

- Accessing the values contained in an object's properties is very simple. You write the name of the variable containing (or referencing) your object, followed by a dot, and then the name of the object's property.
- For example, if you defi ned an Array object contained in the variable myArray, you could access its length property like this:

```
myArray.length
```

- But what can you do with this property now that you have it? You can use it as you would any other piece of data and store it in a variable:

```
var myVariable = myArray.length;
```

- Or you can show it to the user:

```
alert(myArray.length);
```

Primitives

- In JavaScript, the String, Boolean, and Numbers are primitive data types.

Native Objects

- In JavaScript, the String, Array, Date and Math are native objects.
- We can create the String object like the example given below.

```
myVar = new String("Hello how are you?");
myArray = new Array(5);
myDate = new Date( );
```

String Object

- It is the object supported by JavaScript using which we can create string. As we know any object should be created before it is used.
- As we saw we can create the string object like this.

```
var string1 = new String("Hello");
var string2 = new String(123);
var string3 = new String(123.45);
```

Property:

- String object has the length property and it simply returns the number of characters in the string.
- Example:

```
var myvar,var1,var2;
var1 = new String("Hello");
myvar = var1.length;
```

- Here in this example length of the string is 5, so the value 5 will be assigned to the variable myvar.

Methods:

charAt( ) and charCodeAt( )

- If we want to find out information about a single character within a string then we may use charAt( ) and charCodeAt( ) methods.
- The charAt() method accepts one parameter: the index position of the character you want in the string. It then returns that character. charAt() treats the positions of the string characters as starting at 0, so the fi rst character is at index 0, the second at index 1, and so on.

- For example, to find the last character in a string, you could use this code:

```
var myString = prompt("Enter some text","Hello World!");
var theLastChar = myString.charAt(myString.length - 1);
document.write("The last character is " + theLastChar);
```

- Here in this example the character at last position in myString variable will be returned because we have specified myString.length – 1 means the position starts from zero and the length -1 will give you the index of last character.
- The charCodeAt( ) method is similar in use to the charAt( ) method, but instead of returning the Character itself, it returns a number that represents the decimal character code for that character in the Unicode character set. Recall that computers only understand numbers to the computer, all your strings are just numeric data. When you request text rather than numbers, the computer does a conversion based on its internal understanding of each number and provides the respective character.

- For example, to find the character code of the first character in a string, you could write this:

```
var myString = prompt("Enter some text","Hello World!");
var theFirstCharCode = myString.charCodeAt(0);
document.write("The first character code is " + theFirstCharCode);
```

- This will get the character code for the character at index position 0 in the string given by the user, and write it out to the page.
- Character codes go in order, so, for example, the letter A has the code 65, B 66, and so on. Lowercase letters start at 97 (a is 97, b is 98, and so on). Digits go from 48 (for the number 0) to 57 (for the number 9).

indexOf( )

- It is the method supported by string object and this method/function takes two arguments.
  1. The string you want to find.
  2. The character position you want to start searching from (optional). Character positions start from 0 but if you don't include the second parameter, searching starts from 0.
- Syntax:

  Stringname.indexOf("string to be searched","character position to start searching from")
- Example:

```
<script type="text/javascript">
var myString = "Hello jeremy. How are you Jeremy";
var foundAtPosition;
foundAtPosition = myString.indexOf("Jeremy");
alert(foundAtPosition);
</script>
```

- This example code should display a message/alert box containing the number 26 because here we want to search the string "Jeremy". So it would find the last Jeremy word from the main string.

replace( )

This method helps us to replace the part of the string with the other string.

Syntax:

Stringname.replace("the word or regular expression to be searched", "the word or the string to be replaced")

Example:

```
var myvar = "hello how are you?"
var var2=myvar.replace("you","those");
```

Here in this example the main string would be changed and it would be "hello how are those?".

substr( )

By using this method we can create a part of the string i.e. substring.

Syntax:

substr(starting position, length of the substring)

example:

```
var myString = “JavaScript”;
var mySubString = myString.substr(0,4);
alert(mySubString);
```

Here the alert dialog box will display the string “Java” because here the substr( ) function starts to cut from 0th character and it would continue making string up to next 4 characters.

toLowerCase( ) and toUpperCase( )

toLowerCase( ) would convert the whole string to lower case, where as toUpperCase( ) would convert the whole string to Upper case.

Example:

```
var myString = “JavaScript”;
var mylowerstring= myString.toLowerCase( );
var myupperstr = myString.toUpperCase( );
```

toString( )

This method converts the value of variable into the string.

Example:

```
var myInt,myStr;
myInt=345;
myStr = myInt.toString( );
```

Here the value 345 will be converted to the string ”345”.

4. MATH OBJECT

The Math object provides a number of useful mathematical functions and number manipulation methods.

The Math object is a little unusual in that JavaScript automatically creates it for you. There’s no need to declare a variable as a Math object or defi ne a new Math object before being able to use it, making it a little bit easier to use.

The properties of the Math object include some useful math constants, such as the PI property (giving the value 3.14159 and so on). You access these properties, as usual, by placing a dot after the object name (Math) and then writing the property name. For example, to calculate the area of a circle, you may use the following code:

```
var radius = prompt(“Give the radius of the circle”, “”);
var area = Math.PI * radius * radius;
```

```
document.write(“The area is “ + area);
```

The methods of the Math object include some operations that are impossible, or complex, to perform using the standard mathematical operators (+, –, *, and /). For example, the cos() method returns the cosine of the value passed as a parameter.

<u>The abs() Method</u>

The abs() method returns the absolute value of the number passed as its parameter. Essentially, this means that it returns the positive value of the number. So -1 is returned as 1, -4 as 4, and so on. However, 1 would be returned as 1 because it’s already positive.
For example, the following code writes the number 101 to the page.

```
var myNumber = -101;
document.write(Math.abs(myNumber));
```

Finding the Largest and Smallest Numbers: the min() and max() Methods

Let’s say you have two numbers, and you want to fi nd either the largest or smallest of the two. To aid you in this task, the Math object provides the min() and max() methods. These methods both accept at least two arguments, all of which must obviously be numbers. Look at this example code:

```
var max = Math.max(21,22); // result is 22
var min = Math.min(30.1, 30.2); // result is 30.1
```

The min() method returns the number with the lowest value, and max()returns the number with the highest value. The numbers you pass to these two methods can be whole or floating point numbers.
The **max()** *and* **min()** *methods can accept many numbers; you’re not limited to two.*

Rounding Numbers

The Math object provides a few methods to round numbers, each with its own specific purpose.

<u>The ceil() Method</u>

The ceil() method always rounds a number up to the next largest whole number or integer. So 10.01 becomes 11, and –9.99 becomes –9 (because –9 is greater than –10). The ceil() method has just one parameter, namely the number you want rounded up.

Using ceil() is different from using the parseInt() function you saw in Chapter 2, because parseInt()

simply chops off any numbers after the decimal point to leave a whole number, whereas ceil() rounds the number up.

For example, the following code writes two lines in the page, the fi rst containing the number 102 and the second containing the number 101:

```
var myNumber = 101.01;
document.write(Math.ceil(myNumber) + “<br />”);
document.write(parseInt(myNumber));
```

<u>The floor() Method</u>

Like the ceil() method, the floor() method removes any numbers after the decimal point, and returns a whole number or integer. The difference is that floor() always rounds the number down. So if you pass 10.01 you will be returned 10, and if you pass –9.99 you will see –10 returned.

<u>The round() Method</u>

The round() method is very similar to ceil() and floor(), except that instead of always rounding up or always rounding down, it rounds up only if the decimal part is .5 or greater, and rounds down otherwise.

For example:

```
var myNumber = 44.5;
document.write(Math.round(myNumber) + “<br />”);
myNumber = 44.49;
document.write(Math.round(myNumber));
```

This code would write the numbers 45 and 44 to the page.

The **example** given below will give you the idea as a whole.

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0 Transitional//EN”
“http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd”>
<html xmlns=”http://www.w3.org/1999/xhtml”>
<head>
<title>Chapter 5: Example 4</title>
</head>
<body>
<script type=”text/javascript”>
var myNumber = prompt(“Enter the number to be rounded”,““);
```

```
document.write(“<h3>The number you entered was “ + myNumber + “</h3><br />”);
document.write(“<p>The rounding results for this number are</p>”);
document.write(“<table width=’150’ border=’1’>”);
document.write(“<tr><th>Method</th><th>Result</th></tr>”);
document.write(“<tr><td>parseInt()</td><td>”+ parseInt(myNumber) +”</td></tr>”);
document.write(“<tr><td>ceil()</td><td>” + Math.ceil(myNumber) + “</td></tr>”);
document.write(“<tr><td>floor()</td><td>”+ Math.floor(myNumber) + “</td></tr>”);
document.write(“<tr><td>round()</td><td>” + Math.round(myNumber) +”</td></tr>”);
document.write(“</table>”)
</script>
</body>
</html>
```

The first task is to get the number to be rounded from the user.

```
        var myNumber = prompt(“Enter the number to be rounded”,””);
```

Then you write out the number and some descriptive text.

```
document.write(“<h3>The number you entered was “ + myNumber + “</h3><br />”);
document.write(“<p>The rounding results for this number are</p>”);
```

Notice how this time some HTML tags for formatting have been included — the main header being in <h3> tags, and the description of what the table means being inside a paragraph <p> tag.

Next you create the table of results.

```
document.write(“<table width=150 border=1>”);
document.write(“<tr><th>Method</th><th>Result</th></tr>”);
document.write(“<tr><td>parseInt()</td><td>”+ parseInt(myNumber) +”</td></tr>”);
document.write(“<tr><td>ceil()</td><td>” + Math.ceil(myNumber) + “</td></tr>”);
document.write(“<tr><td>floor()</td><td>”+ Math.floor(myNumber) + “</td></tr>”);
document.write(“<tr><td>round()</td><td>” + Math.round(myNumber) +”</td></tr>”);
document.write(“</table>”)
```

You create the table header first before actually displaying the results of each rounding function on a separate row. You can see how easy it is to dynamically create HTML inside the web page using just JavaScript. The principles are the same as with HTML in a page: You must make sure your tag’s syntax is valid or otherwise things will appear strange or not appear at all.
Each row follows the same principle but uses a different rounding function. Let’s look at the fi rst

row, which displays the results of parseInt().

```
document.write("<tr><td>parseInt()</td><td>"+ parseInt(myNumber) +"</td></tr>");
```

Inside the string to be written out to the page, you start by creating the table row with the <tr> tag. Then you create a table cell with a <td> tag and insert the name of the method from which the results are being displayed on this row. Then you close the cell with </td> and open a new one with <td>.

Inside this next cell you are placing the actual results of the parseInt() function. Although a number is returned by parseInt(), because you are concatenating it to a string, JavaScript automatically converts the number returned by parseInt() into a string before concatenating. All this happens in the background without you needing to do a thing. Finally, you close the cell and the row with </td></tr>.

The pow() Method

The pow() method raises a number to a specifi ed power. It takes two parameters, the fi rst being the number you want raised to a power, and the second being the power itself. For example, to raise 2 to the power of 8 (that is, to calculate 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2), you would write Math.pow(2,8) — the result being 256. Unlike some of the other mathematical methods, like sin(), cos(), and acos(), which are not commonly used in web programming unless it's a scientifi c application you're writing, the pow() method can often prove very useful.

5. DATE / TIME

The Date object handles everything to do with date and time in JavaScript. Using it, you can find out the current date and time, store your own dates and times, do calculations with these dates, and convert the dates into strings.

The Date object has a lot of methods and can be a little tricky to use, which is why Chapter 10 is dedicated to the date, time, and timers in JavaScript. You'll also see in Chapter 12 how you can use dates to determine if there's been anything new added to the web site since the user last visited it. However, in this section you'll focus on how to create a Date object and some of its more commonly used methods.

Creating a Date Object

You can declare and initialize a Date object in four ways. In the fi rst method, you simply declare a new **Date** object without initializing its value. In this case, the date and time value will be set to the current date and time on the PC on which the script is run.

```
var theDate1 = new Date();
```

Secondly, you can defi ne a Date object by passing the number of milliseconds since January 1, 1970, at 00:00:00 GMT. In the following example, the date is 31 January 2000 00:20:00 GMT (that is, 20 minutes past midnight).

```
var theDate2 = new Date(949278000000);
```

This is how JavaScript actually stores the dates. The other formats for giving a date are simply for convenience. Next, you can pass a string representing a date, or a date and time. In the following example, you have “31 January 2010”.

```
var theDate3 = new Date(“31 January 2010”);
```

However, you could have written 31 Jan 2010, Jan 31 2010, or any of a number of valid variations you’d commonly expect when writing down a date normally — if in doubt, try it out. Note that Firefox doesn’t support the string “01-31-2010” as a valid date format.
If you are writing your web pages for an international audience outside the United States, you need to be aware of the different ways of specifying dates. In the United Kingdom and many other places, the standard is day, month, year, whereas in the United States the standard is month, day, year. This can cause problems if you specify only numbers — JavaScript may think you’re referring to a day when you mean a month. The easiest way to avoid such headaches is to, where possible, always use the name of
the month. That way there can be no confusion.

In the fourth and final way of defining a Date object, you initialize it by passing the following parameters separated by commas: year, month, day, hours, minutes, seconds, and milliseconds.
For example:

```
var theDate4 = new Date(2010,0,31,15,35,20,20);
```

This date is actually 31 January 2010 at 15:35:20 and 20 milliseconds. You can specify just the date part if you wish and ignore the time. Something to be aware of is that in this instance January is month 0, not month 1, as you’d expect, and
December is month 11. It’s very easy to make a mistake when specifying a month.

Getting Date Values

It’s all very nice having stored a date, but how do you get the information out again? Well, you just use
the get methods. These are summarized in the following table.

Method Returns

getDate()	The day of the month
getDay()	The day of the week as an integer, with Sunday as 0, Monday as 1, and so on
getMonth()	The month as an integer, with January as 0, February as 1, and so on
getFullYear()	The year as a four-digit number
toDateString()	Returns the full date based on the current time zone as a human-readable

Returns the full date based on the current time zone as a human-readable string. For example, “Wed 31 Dec 2003”. For example, if you want to get the month in ourDateObj, you can simply write the following:

```
theMonth = myDateObject.getMonth();
```

All the methods work in a very similar way, and all values returned are based on local time, meaning time local to the machine the code is running on. It’s also possible to use Universal Time, previously known as GMT.

Using the Date Object to Retrieve the Current Date In this example, you use the get date type methods you have been looking at to write the current day, month, and year to a web page.

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0 Transitional//EN”
“http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd”>
<html xmlns=”http://www.w3.org/1999/xhtml”>
<head>
<title>Chapter 5: Example 6</title>
</head>
<body>
<script type=”text/javascript”>
var months = new Array(“January”,”February”,”March”,”April”,”May”,”June”,”July”,
“August”, “September”, “October”, “November”, “December”);
var dateNow = new Date();
var yearNow = dateNow.getFullYear();
var monthNow = months[dateNow.getMonth()];
var dayNow = dateNow.getDate();
var daySuffix;
switch (dayNow)
{
```

```
case 1:
case 21:
case 31:
daySuffix = “st”;
break;
case 2:
case 22:
daySuffix = “nd”;
break;
case 3:
case 23:
daySuffix = “rd”;
break;
default:
daySuffix = “th”;
break;
}
document.write(“It is the “ + dayNow + daySuffix + “ day “);
document.write(“in the month of “ + monthNow);
document.write(“ in the year “ + yearNow);
</script>
</body>
</html>
```

If you load up the page, you should see a correctly formatted sentence telling you what the current date is. The first thing you do in the code is declare an array and populate it with the months of a year. Why do this? Well, there is no method of the Date object that’ll give you the month by name instead of as a number. However, this poses no problem; you just declare an array of months and use the month number as the array index to select the correct month name.

```
var months = new Array(“January”,”February”,”March”,”April”,”May”,”June”,”July”,
“August”,”September”,”October”,”November”,”December”);
```

Next you create a new Date object, and by not initializing it with your own value, you allow it to initialize itself to the current date and time.

```
var dateNow = new Date();
```

Following this you set the yearNow variable to the current year, as returned by the getFullYear()

method.

```
var yearNow = dateNow.getFullYear();
```

You then populate your monthNow variable with the value contained in the array element with an index of the number returned by getMonth(). Remember that getMonth() returns the month as an integer value, starting with 0 for January — this is a bonus because arrays also start at 0, so no adjustment is needed to fi nd the correct array element.

```
var monthNow = months[dateNow.getMonth()];
```

Finally, the current day of the month is put into variable dayNow.

```
var dayNow = dateNow.getDate();
```

Next you use a switch statement, which you learned about in the Chapter 3. This is a useful technique for adding the correct suffix to the date that you already have. After all, your application will look more Professional if you can say “it is the 1st day”, rather than “it is the 1 day”. This is a little tricky, however, because the suffix you want to add depends on the number that precedes it. So, for the first, twenty-first, and thirty-first days of the month, you have this:

```
switch (dayNow)
{
case 1:
case 21:
case 31:
daySuffix = “st”;
break;
```

For the second and twenty-second days, you have this:

```
case 2:
case 22:
daySuffix = “nd”;
break;
```

and for the third and twenty-third days, you have this:

```
case 3:
case 23:
```

```
daySuffix = “rd”;
break;
```

Finally, you need the default case for everything else. As you will have guessed by now, this is simply “th”.

```
default:
daySuffix = “th”;
break;
}
```

In the final lines you simply write the information to the HTML page, using document.write().

Setting Date Values

To change part of the date in a Date object, you have a group of set functions, which pretty much replicate the get functions described earlier, except that you are setting, not getting, the values. These functions are summarized in the following table.

Method	**Description**
setDate()	The date of the month is passed in as the parameter to set the date
setMonth()	The month of the year is passed in as an integer parameter, where 0 is January, 1 is February, and so on
setFullYear()	This sets the year to the four-digit integer number passed in as a parameter

Note that for security reasons, there is no way for web-based JavaScript to change the current date and time on a user’s computer.

So, to change the year to 2009, the code would be as follows:

```
myDateObject.setFullYear(2009);
```

Setting the date and month to the twenty-seventh of February looks like this:

```
myDateObject.setDate(27);
myDateObject.setMonth(1);
```

One minor point to note here is that there is no direct equivalent of the getDay() method. After the year, date, and month have been defi ned, the day is automatically set for you.

Calculations and Dates

Take a look at the following code:

```
var myDate = new Date(“1 Jan 2010”);
myDate.setDate(32);
document.write(myDate);
```

Surely there is some error — since when has January had 32 days? The answer is that of course it doesn’t, and JavaScript knows that. Instead JavaScript sets the date to 32 days from the fi rst of January — that is, it sets it to the fi rst of February.

The same also applies to the setMonth() method. If you set it to a value greater than 11, the date automatically rolls over to the next year. So if you use setMonth(12), that will set the date to January of the next year, and similarly setMonth(13) is February of the next year.

How can you use this feature of setDate() and setMonth() to your advantage? Well, let’s say you want to find out what date it will be 28 days from now. Given that different months have different numbers of days and that you could roll over to a different year, it’s not as simple a task as it might fi rst seem. Or at least that would be the case if it were not for setDate(). The code to achieve this task is as follows:

```
var nowDate = new Date();
var currentDay = nowDate.getDate();
nowDate.setDate(currentDay + 28);
```

First you get the current system date by setting the nowDate variable to a new Date object with no initialization value. In the next line, you put the current day of the month into a variable called currentDay. Why? Well, when you use setDate() and pass it a value outside of the maximum number of days for that month, it starts from the fi rst of the month and counts that many days forward. So, if today’s date is the January 15 and you use setDate(28), it’s not 28 days from the fi fteenth of January, but 28 days from the fi rst of January. What you want is 28 days from the current date, so you need to add the current
date to the number of days ahead you want. So you want setDate(15 + 28). In the third line, you set the date to the current date, plus 28 days. You stored the current day of the month in currentDay, so now you just add 28 to that to move 28 days ahead.

If you want the date 28 days prior to the current date, you just pass the current date minus 28. Note that this will most often be a negative number. You need to change only one line, and that’s the third one, which you change to the following:

```
nowDate.setDate(currentDay - 28);
```

You can use exactly the same principles for setMonth() as you have used for setDate().

Getting Time Values

The methods you use to retrieve the individual pieces of time data work much like the get methods for date values. The methods you use here are:

❑ getHours()
❑ getMinutes()
❑ getSeconds()
❑ getMilliseconds()
❑ toTimeString()

These methods return respectively the hours, minutes, seconds, milliseconds, and full time of the specifi ed Date object, where the time is based on the 24-hour clock: 0 for midnight and 23 for 11 p.m. The last method is similar to the toDateString() method in that it returns an easily readable string, except that in this case it contains the time (for example, “13:03:51 UTC”).

Example: <u>Writing the Current Time into a Web Page</u>

Let’s look at an example that writes out the current time to the page.

```
<!DOCTYPE html PUBLIC “-//W3C//DTD XHTML 1.0 Transitional//EN”
“http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd”>
<html xmlns=”http://www.w3.org/1999/xhtml”>
<head>
<title>Chapter 5: Example 7</title>
</head>
<body>
<script type=”text/javascript”>
var greeting;
var nowDate = new Date();
var nowHour = nowDate.getHours();
var nowMinute = nowDate.getMinutes();
var nowSecond = nowDate.getSeconds();
if (nowMinute < 10)
{
nowMinute = “0” + nowMinute;
}
if (nowSecond < 10)
{
nowSecond = “0” + nowSecond;
}
if (nowHour < 12)
{
```

```
greeting = “Good Morning”;
}
else if (nowHour < 17)
{
greeting = “Good Afternoon”;
}
else
{
greeting = “Good Evening”;
}
document.write(“<h4>” + greeting + “ and welcome to my website</h4>”)
document.write(“According to your clock the time is “);
document.write(nowHour + “:” + nowMinute + “:” + nowSecond);
</script>
</body>
</html>
```

Save this page as yourfilename.htm. When you load it into a web browser, it writes a greeting based on the time of day as well as the current time.

The fi rst two lines of code declare two variables — greeting and nowDate.

```
var greeting;

var nowDate = new Date();
```

The greeting variable will be used shortly to store the welcome message on the web site, whether this is “Good Morning”, “Good Afternoon”, or “Good Evening”. The nowDate variable is initialized to a new Date object. Note that the constructor for the Date object is empty, so JavaScript will store the current date and time in it.

Next, you get the information on the current time from nowDate and store it in various variables. You can see that getting time data is very similar to getting date data, just using different methods.

```
var nowHour = nowDate.getHours();
var nowMinute = nowDate.getMinutes();
var nowSecond = nowDate.getSeconds();
```

You may wonder why the following lines are included in the example:

```
if (nowMinute < 10)
{
```

```
nowMinute = “0” + nowMinute;
}
if (nowSecond < 10)
{
nowSecond = “0” + nowSecond;
}
```

These lines are there just for formatting reasons. If the time is nine minutes past 10, then you expect to see something like 10:09. You don’t expect 10:9, which is what you would get if you used the getMinutes() method without adding the extra zero. The same goes for seconds. If you’re just using the data in calculations, you don’t need to worry about formatting issues — you do here because you’re inserting the time the code executed into the web page. Next, in a series of if statements, you decide (based on the time of day) which greeting to create for displaying to the user.

```
if (nowHour < 12)
{
greeting = “Good Morning”;
}
else if (nowHour < 17)
{
greeting = “Good Afternoon”;
}
else
{
greeting = “Good Evening”;
}
```

Finally, you write out the greeting and the current time to the page.

```
document.write(“<h4>” + greeting + “ and welcome to my website</h4>”);
document.write(“According to your clock the time is “);
document.write(nowHour + “:” + nowMinute + “:” + nowSecond);
```

Setting Time Values

When you want to set the time in your Date objects, you have a series of methods similar to those used for getting the time:

❑ setHours()
❑ setMinutes()

❑ setSeconds()
❑ setMilliseconds()

These work much like the methods you use to set the date, in that if you set any of the time parameters to an illegal value, JavaScript assumes you mean the next or previous time boundary. If it's 9:57 and you set minutes to 64, the time will be set to 10:04 — that is, 64 minutes from 9:00.

This is demonstrated in the following code:

```
var nowDate = new Date();
nowDate.setHours(9);
nowDate.setMinutes(57);
alert(nowDate);
nowDate.setMinutes(64);
alert(nowDate);
```

First you declare the nowDate variable and assign it to a new Date object, which will contain the current date and time. In the following two lines, you set the hours to 9 and the minutes to 57. You show the date and time using an alert box, which should show a time of 9:57. The minutes are then set to 64 and again an alert box is used to show the date and time to the user. Now the minutes have rolled over the hour so the time shown should be 10:04.

If the hours were set to 23 instead of 9, setting the minutes to 64 would not just move the time to another hour but also cause the day to change to the next date.

UNIT-IV: LONG QUESTIONS

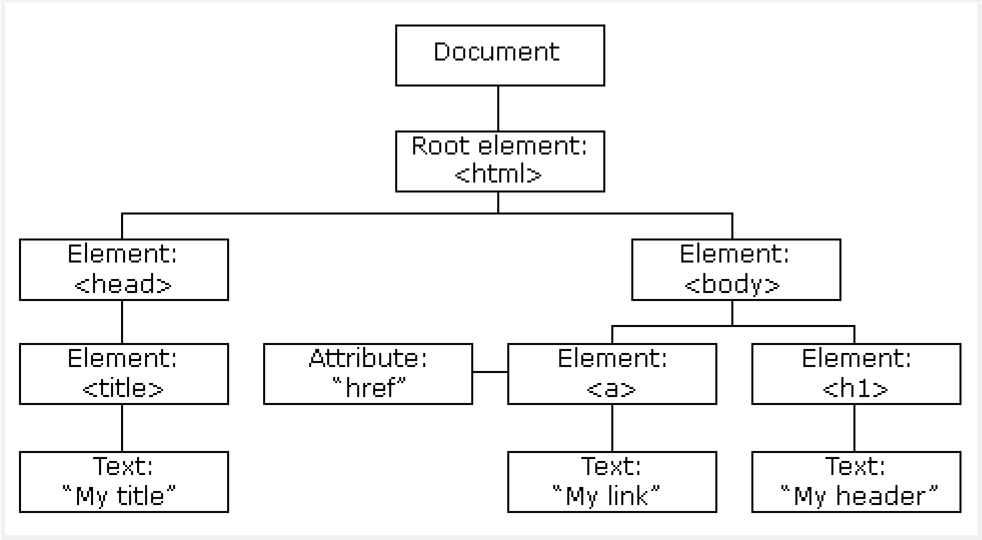
1. **Draw and discuss the structure of DOM**

The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.

The **HTML DOM** model is constructed as a tree of **Objects**:

The HTML DOM Tree of Objects



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

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

What is the DOM?

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The W3C DOM standard is separated into 3 different parts:

- Core DOM - standard model for all document types
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

What is the HTML DOM?

The HTML DOM is a standard **object** model and **programming interface** for HTML. It defines:

- The HTML elements as **objects**
- The **properties** of all HTML elements
- The **methods** to access all HTML elements
- The **events** for all HTML elements

In other words: **The HTML DOM is a standard for how to get, change, add, or delete HTML elements.**

2. **<u>What is DOM Hierarchy? Explain.</u>**

The Document Object Model (DOM) is a way of representing the document independent of browser type. It allows a developer to access the document via a common set of objects, properties, methods, and events, and to alter the contents of the web page dynamically using scripts. Several types of script languages, such as JavaScript and VBScript, are available. Each requires a different syntax and therefore a different approach when you’re programming. Even when you’re using a language common to all browsers, such as JavaScript, you should be aware that some small variations are usually added to the language by the browser vendor.

What gives JavaScript this power over a web page is the *Document Object Model (DOM)*, a tree-like representation of the web page.

The DOM is one of the standards set forth by the World Wide Web Consortium (W3C), a body of developers who recommend standards for browser makers and web developers to follow. The DOM gives developers a way of representing everything on a web page so that it is accessible via a common set of properties and methods in JavaScript. You can literally change anything on the page: the graphics, tables, forms, and even text itself by altering a relevant DOM property with JavaScript.

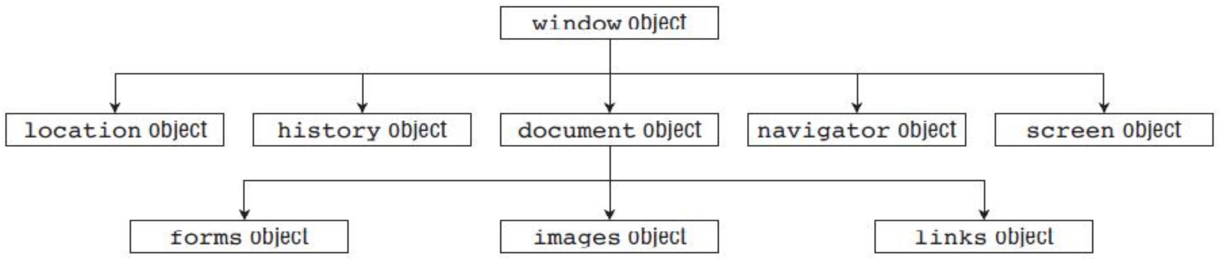
The DOM should not be confused with the Browser Object Model (BOM). For now, though, think of the BOM as a browser dependent representation of every feature of the browser, from the browser buttons, URL address line, and title bar to the browser window controls, as well as parts of the web page, too.

The DOM, however, deals only with the contents of the browser window or web page (in other words, the HTML document). It makes the document available in such a way that any browser can use exactly the same code to access and manipulate the content of the document. In short, the BOM gives you access to the browser and some of the document, whereas the DOM gives you access to all of the document, but *only* the document.

The great thing about the DOM is that it is browser- and platform-independent. This means that developers can finally consider the possibility of writing a piece of JavaScript code that dynamically updates the page, and that will work on any DOM-compliant browser without any tweaking. You should not need to code for different browsers or take excessive care when coding.

DOM Hierarchy

The Document Object Model (DOM) hierarchy may look like the following:



As we can logically think of the DOM hierarchy like the window as the outer part of the browser, here also in the hierarchy ***window*** object is lied at the top. Other objects lie under the ***window*** object e.g. ***location*** object, ***history*** object, ***document*** object, ***navigator*** object etc.

3. **Explain Window object in detail.**

The window object represents the browser's **frame** or **window**, in which your web page is contained. To some extent, it also represents the browser itself and includes a number of properties that are there simply because they don't fit anywhere else. For example, via the properties of the window object, you can find out what browser is running, the pages the user has visited, the size of the browser window, the size of the user's screen, and much more. You can also use the window object to access and change the text in the browser's status bar, change the page that is loaded, and even open new windows.

The window object is a ***global*** *object*, which means you don't need to use its name to access its properties and methods. In fact, the global functions and variables (the ones accessible to script anywhere in a page) are all created as properties of the global object. For example, the alert() function you have been using since the beginning of the book is, in fact, the alert() method of the window object. Although you have been using this simply as this:

```
alert("Hello!");
```

You could write this with the same, exact results:

```
window.alert("Hello!");
```

However, since the window object is the global object, it is perfectly correct to use the first version.

Some of the properties of the window object are themselves objects. Those common to all browsers include the document, navigator, history, screen, and location objects.

The document object represents your page, the history object contains the history of pages visited by the user, the navigator object holds information about the browser, the screen object contains information about the display capabilities of the client, and the location object contains details on the current page's location. Let's start with a nice, simple example in which you change the default text shown in the browser's status bar. The status bar (usually in the bottom left of the browser window) is usually used by the browser to show the status of any document loading into the browser. For example, on IE and Firefox, after a document has loaded, you'll normally see Done in the status bar. Let's change that so it says "Hello and Welcome."

To change the default message in the window's status bar, you need to use the window object's defaultStatus property. To do this, you can write the following:

```
window.defaultStatus = "Hello and Welcome";
```

Or, because the window is the global object, you can just write this:

```
defaultStatus = "Hello and Welcome";
```

4. **<u>Explain Document Object in detail.</u>**

Along with the window object, the document object is probably one of the most important and commonly used objects in the BOM. Via this object you can gain access to the HTML elements, their properties and methods inside your page.

Unfortunately, it's here, at the document object, that browsers can differ greatly. This chapter concentrates on the properties and methods that are common to all browsers.

The document object has a number of properties associated with it, which are also array-like structures called *collections*. The main collections are the forms, images, and links collections. IE supports a number of other collection properties, such as the all collection property, which is an array of all the elements represented by objects in the page. However, you'll be concentrating on using objects that have cross-browser support, so that you are not limiting your web pages to just one browser.

You'll be looking at the images and links collections shortly. A third collection, the forms collection, will be one of the topics of the next chapter when you look at forms in web browsers. First, though, you'll look at a nice, simple example of how to use the document object's methods and properties.

<u>Using the document Object</u>

You've already come across some of the document object's properties and methods, for example the write() method and the bgColor property.

<u>Example</u>:

Setting Colors According to the User's Screen Color Depth

In this example, you set the background color of the page according to how many colors the user's screen supports. This is termed *screen color depth*. If the user has a display that supports just two colors (black and white), there's no point in you setting the background color to bright red. You accommodate different depths by using JavaScript to set a color the user can actually see.

```
<html>
<head>
<title>setting colors according to the user’s screen color depth</title>
</head>
<body>
<script type=”text/javaScript”>
switch (window.screen.colorDepth) {
case 1:
case 4:
document.bgColor = “white”;
break;
case 8:
case 15:
case 16:
document.bgColor = “blue”;
break;
case 24:
case 32:
document.bgColor = “skyblue”;
break;
default:
document.bgColor = “white”;
}
document.write(“Your screen supports “ + window.screen.colorDepth + “bit color”);
</script>
</body>
</html>
```

Save the page as **setting_colors.html**. When you load it into your browser, the background color of the page will be determined by your current screen color depth. Also, a message in the page will tell you what the color depth currently is.

As you saw earlier, the window object has the screen object property. One of the properties of this object is the **colorDepth** property, which returns a value of **1, 4, 8, 15, 16, 24, or 32**. This represents the number of bits assigned to each pixel on your screen. (A pixel is just one of the many dots that your screen is made up of.)

To work out how many colors you have, you just calculate the value of 2 to the power of the **colorDepth** property. For example, a **colorDepth** of 1 means that there are two colors available, a **colorDepth** of 8 means that there are 256 colors available, and so on. Currently, most people have a screen color depth of at least 8, but usually 24 or 32.

The first task of the script block is to set the color of the background of the page based on the number of colors the user can actually see. You do this in a big switch statement. The condition that is checked for in the switch statement is the value of window.screen.colorDepth.

```
switch (window.screen.colorDepth)
```

You don't need to set a different color for each colorDepth possible, because many of them are similar when it comes to general web use. Instead, you set the same background color for different, but similar, **colorDepth** values. For a **colorDepth** of 1 or 4, you set the background to white. You do this by declaring the case 1: statement, but you don't give it any code.

If the **colorDepth** matches this case statement, it will fall through to the case 4: statement below, where you do set the background color to white. You then call a break statement, so that the case matching will not fall any further through the switch statement.

```
{
case 1:
case 4:
document.bgColor = "white";
break;
```

You do the same with **colorDepth** values of 8, 15, and 16, setting the background color to blue as follows:

```
case 8:
case 15:
case 16:
document.bgColor = "blue";
break;
```

Finally, you do the same for **colorDepth** values of 24 and 32, setting the background color to sky blue.

```
case 24:
case 32:
document.bgColor = "skyblue";
break;
```

You end the switch statement with a default case, just in case the other case statements did not match. In this default case, you again set the background color to white.

```
default:
document.bgColor = "white";
}
```

In the next bit of script, you use the document object's write() method, something you've been using in these examples for a while now. You use it to write to the document — that is, the page — the number of bits the color depth is currently set at, as follows:

```
document.write("Your screen supports " + window.screen.colorDepth + "bit color")
```

5. Explain History Object in detail.

The history object keeps track of each page that the user visits. This list of pages is commonly called the *history stack* for the browser. It enables the user to click the browser's Back and Forward buttons to revisit pages. You have access to this object via the window object's history property.

Like the native JavaScript Array type, the history object has a length property. You can use this to find out how many pages are in the history stack.

As you might expect, the history object has the **back()** and **forward()** methods. When they are called, the location of the page currently loaded in the browser is changed to the previous or next page that the user has visited. The history object also has the **go()** method. This takes one parameter that specifies how far forward or backward in the history stack you want to go.

For example, if you wanted to return the user to the page before the previous page, you'd write this:

```
history.go(-2);
```

To go forward three pages, you'd write this:

```
history.go(3);.
```

Note that go(-1) and back() are equivalent, as are go(1) and forward().

6. Explain Location Object in detail.

The location object contains lots of potentially useful information about the current page's location. Not only does it contain the **Uniform Resource Locator** (URL) for the page, but also the server hosting the page, the port number of the server connection, and the protocol used. This information is made available through the location object's **href**, hostname, port, and protocol properties. However, many of these values are only really relevant when you are loading the page from a server and not, as you are doing in the present examples, loading the page directly from a

local hard drive. In addition to retrieving the current page's location, you can use the methods of the location object to change the location and refresh the current page.
You can navigate to another page in two ways. You can either set the location object's **href** property to point to another page, or you can use the location object's **replace()** method. The effect of the two is the same; the page changes location. However, they differ in that the **replace()** method removes the current page from the history stack and replaces it with the new page you are moving to, whereas using the **href** property simply adds the new page to the top of the history stack. This means that if the **replace()** method has been used and the user clicks the Back button in the browser, the user can't go back to the original page loaded. If the **href** property has been used, the user can use the Back button as normal.

For example, to replace the current page with a new page called myPage.htm, you'd use the **replace()** method and write the following:

```
window.location.replace("myPage.htm");
```

This will load myPage.htm and replace any occurrence of the current page in the history stack with myPage.htm. To load the same page and to add it to the history of pages navigated to, you use the **href** property:

```
window.location.href = "myPage.htm";
```

and the page currently loaded is added to the history. In both of the preceding cases, window is in front of the expression, but as the window object is global throughout the page, you could have written one of the following:

```
location.replace("myPage.htm");
location.href = "myPage.htm";
```

7. <u>Explain forms collection in detail.</u>

Forms provide you with a way of grouping together HTML interaction elements with a common purpose. For example, a form may contain elements that enable the input of a user's data for registering on a web site. Another form may contain elements that enable the user to ask for a car insurance quote. It's possible to have a number of separate forms in a single page. You don't need to
worry about pages containing multiple forms until you have to submit information to a web server — then you need to be aware that the information from only one of the forms on a page can be submitted to the server at one time.

To create a form, use the <form> and </form> tags to declare where it starts and where it ends. The <form/> element has a number of attributes, such as the` action attribute, which determines where the form is submitted to; the method attribute, which determines how the information is submitted; and the target attribute, which determines the frame to which the response to the form is loaded.

Generally speaking, for client-side scripting where you have no intention of submitting information to a server, these attributes are not necessary. They will come into play in a later chapter when you look at programming server pages. For now the only attribute you need to set in the <form/> element is the name attribute, so that you can reference the form.

So, to create a blank form, the tags required would look something like this:

```
        <form name=”myForm”>
        </form>
```

First, you can access the object directly using its name — in this case document.myForm. Alternatively, you can access the object through the document object’s forms collection property. Remember that the last chapter included a discussion of the document object’s images collection and how you can manipulate it like any other array.

The same applies to the forms collection, except that instead of each element in the collection holding an IMG object, it now holds a Form object. For example, if it’s the first Form in the page, you reference it using document.forms[0].

Many of the attributes of the <form/> element can be accessed as properties of the Form object. In particular, the name property of the Form object mirrors the name attribute of the <form/> element.

Example

Let’s have a look at an example that uses the forms collection. Here you have a page with three forms on it. Using the forms collection, you access each Form object in turn and show the value of its name property in a message box.

```
<html >
<head>
<title>Forms collection</title>
<script type=”text/javascript”>
function window_onload()
{
var numberForms = document.forms.length;
var Index;
for (Index = 0; Index < numberForms; Index++)
{        alert(document.forms[Index].name);          }
}
</script>
</head>
<body onload=”window_onload()“>
        <form action=”“ name=”form1”>
        <p>
        This is inside form1.
        </p>
        </form>
        <form action="“ name="form2">
        <p>
        This is inside form2
        </p>
        </form>
        <form action="“ name="form3">
        <p>
        This is inside form3
        </p>
        </form>
</body>
</html>
```

Save this as **Forms_collection.html**. When you load it into your browser, you should see three alert boxes, each of which shows the name of a form. Within the body of the page you define three forms. Each form is given a name and contains a paragraph of text.

Within the definition of the <body/> element, the window_onload() function is connected to the window object's onload event handler.

```
        <body onload="window_onload()“>
```

This means that when the page is loaded, your window_onload() function will be called. The window_onload() function is defined in a script block in the head of the page. Within this function you loop through the forms collection. Just like any other JavaScript array, the forms collection has a length property, which you can use to determine how many times you need to loop. Actually, because you know how many forms there are, you can just write the number in. However, this example uses the length property, since that makes it easier to add to the collection without having to change the function. Generalizing your code like this is a good practice to get into.

The function starts by getting the number of Form objects within the forms array and storing that number in the variable numberForms.

```
function window_onload()
{
var numberForms = document.forms.length;
```

Next you define a variable, formIndex, to be used in the for loop.

```
var formIndex;
```

```
for (formIndex = 0; formIndex < numberForms; formIndex++)
{
alert(document.forms[formIndex].name);
}
}
```

Remember that because the indexes for arrays start at 0, your loop needs to go from an index of 0 to an index of numberForms – 1. You enable this by initializing the formIndex variable to 0, and setting the condition of the for loop to formIndex < numberForms. Within the for loop's code, you pass the index of the form you want (that is, formIndex) to document .forms[], which gives you the Form object at that index in the forms collection. To access the Form object's name property, you put a dot at the end of the name of the property, name.

9. **<u>Explain links collection in detail.</u>**

For each hyperlink element <a/> defined with an **href** attribute, the browser creates an a object. The most important property of the a object is the **href** property, corresponding to the **href** attribute of the tag. Using this, you can find out where the link points to, and you can change this even after the page has loaded.
The collection of all a objects in a page is contained within the links collection, much as the img objects are contained in the images collection, as you saw earlier.

10. **<u>Explain images collection in detail.</u>**

As you know, you can insert an image into an HTML page using the following tag:

```
<img alt="USA" name="myImage" src="usa.gif" />
```

The browser makes this image available for you to manipulate with JavaScript by creating an img object for it with the name myImage. In fact, each image on your page has an img object created for it.

Each of the **img** objects in a page is stored in the images collection, which is a property of the document object. You use this, and other collections, as you would an array. The first image on the page is found in the element **document.images[0]**, the second in **document.images[1]**, and so on.
If you want to, you can assign a variable to reference an img object in the images collection. It can make code easier to read. For example, the following code assigns a reference to the img object at index position 1 to the myImage2 variable:

```
var myImage2 = document.images[1];
```

Now you can write **myImage2** instead of **document.images[1]** in your code, with exactly the same effect.

You can also access **img** objects in the images collection by name. For example, the **img** object created by the <**img**/> element, which has the name **myImage**, can be accessed in the document object's images collection property like this:

```
document.images["myImage"]
```

Because the **document.images** property is a collection, it has the properties similar to the native JavaScript Array type, such as the length property. For example, if you want to know how many images there are on the page, the code **document.images.length** will tell you.

Example: Image Selection

The img object itself has a number of useful properties. The most important of these is its src property. By changing this, you can change the image that's loaded. The next example demonstrates this.

```
<html>
<head>
<title>Chapter 5: Image selection</title>
</head>
<body>
<img name="img1" src="" border="0" width="200" height="150" />
<script type="text/javaScript">
var myImages = new Array("usa.gif","canada.gif","jamaica.gif","mexico.gif");
var imgIndex = prompt("Enter a number from 0 to 3","");
document.images["img1"].src = myImages[imgIndex];
</script>
</body>
</html>
```

Save this as **ImgCollection.html**. You will also need four image files, called **usa.gif, canada.gif, jamaica.gif,** and **mexico.gif**.

A prompt box asks you to enter a number from 0 to 3 when this page loads into the browser. A different image will be displayed depending on the number you enter.

At the top of the page you have your HTML <img/> element. Notice that the src attribute is left empty and is given the name value img1.

```
<img name="img1" src="" border="0" width="200" height="150">
```

Next you come to the script block where the image to be displayed is decided. On the first line, you define an array containing a list of image sources. In this example, the images are in the same directory as the HTML file, so a path is not specified. If yours are not, make sure you enter the full path (for example, C:\myImages\mexico.gif).

Then you ask the user for a number from 0 to 3, which will be used as the array index to access the image source in the myImages array.

    var imgIndex = prompt("Enter a number from 0 to 3","");

Finally, you set the src property of the img object to the source text inside the myImages array element with the index number provided by the user.

    document.images["img1"].src = myImages[imgIndex];

Don't forget that when you write document.images["img1"], you are accessing the img object stored

in the images collection. You've used the image's name, as defined in the name attribute of the <img/> element, but you could have used document.images[0]. It's an index position of 0, because it's the first (and only) image on this page.

11. **What is Event? What is Event Handling? How you handle the event – explain with an example.**

HTML events are **"things"** that happen to HTML elements.
When JavaScript is used in HTML pages, JavaScript can **"react"** on these events.
HTML Events
An HTML event can be something the browser does, or something a user does.

Here are some examples of HTML events:

- An HTML web page has finished loading
- An HTML input field was changed
- An HTML button was clicked

Often, when events happen, you may want to do something.
JavaScript lets you execute code when events are detected.
HTML allows event handler attributes, **with JavaScript code**, to be added to HTML elements.

A JavaScript can be executed when an event occurs, like when a user clicks on an HTML element.
To execute code when a user clicks on an element, add JavaScript code to an HTML event attribute:
onclick=*JavaScript*
Examples of HTML events:

Event	Description
onchange	An HTML element has been changed

Onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

Handling the event

The onclick event

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

<h1 onclick="this.innerHTML = 'Ooops!'">Click on this text!</h1>

</body>
</html>
```

The onload and onunload Events

The onload and onunload events are triggered when the user enters or leaves the page.

The onload event can be used to check the visitor's browser type and browser version, and load the proper version of the web page based on the information.

The onload and onunload events can be used to deal with cookies.

Example

```
<body onload="checkCookies()">
```

The onchange Event

The onchange event is often used in combination with validation of input fields.

Below is an example of how to use the onchange. The upperCase() function will be called when a user changes the content of an input field.

Example

```
<input type="text" id="fname" onchange="upperCase()">
```

The onmouseover and onmouseout Events

The onmouseover and onmouseout events can be used to trigger a function when the user mouses over, or out of, an HTML element:

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

<div onmouseover="mOver(this)" onmouseout="mOut(this)"
style="background-color:#D94A38;width:120px;height:20px;padding:40px;">
Mouse Over Me</div>

<script>
function mOver(obj) {
   obj.innerHTML = "Thank You"
}

function mOut(obj) {
   obj.innerHTML = "Mouse Over Me"
}
</script>

</body>
</html>
```

The onmousedown, onmouseup and onclick Events

The onmousedown, onmouseup, and onclick events are all parts of a mouse-click. First when a mouse-button is clicked, the onmousedown event is triggered, then, when the mouse-button is released, the onmouseup event is triggered, finally, when the mouse-click is completed, the onclick event is triggered.

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

<div onmousedown="mDown(this)" onmouseup="mUp(this)"
style="background-color:#D94A38;width:90px;height:20px;padding:40px;">
Click Me</div>

<script>
function mDown(obj) {
   obj.style.backgroundColor = "#1ec5e5";
```

```
  obj.innerHTML = "Release Me";
}

function mUp(obj) {
  obj.style.backgroundColor="#D94A38";
  obj.innerHTML="Thank You";
}
</script>

</body>
</html>
```

Keyboard Events

Event	Description
onkeydown	The event occurs when the user is pressing a key
onkeypress	The event occurs when the user presses a key
onkeyup	The event occurs when the user releases a key

Mouse Events

Event	Description
onclick	The event occurs when the user clicks on an element
oncontextmenu	The event occurs when the user right-clicks on an element to open a context menu
ondblclick	The event occurs when the user double-clicks on an element
onmousedown	The event occurs when the user presses a mouse button over an element
onmouseenter	The event occurs when the pointer is moved onto an element
onmouseleave	The event occurs when the pointer is moved out of an element
onmousemove	The event occurs when the pointer is moving while it is over an element
onmouseover	The event occurs when the pointer is moved onto an element, or onto one of its children
onmouseout	The event occurs when a user moves the mouse pointer out of an element, or out of one of its children
onmouseup	The event occurs when a user releases a mouse button over an element

Form Events

Event	Description
onblur	The event occurs when an element loses focus
onchange	The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>)
onfocus	The event occurs when an element gets focus
onfocusin	The event occurs when an element is about to get focus
onfocusout	The event occurs when an element is about to lose focus
oninput	The event occurs when an element gets user input
oninvalid	The event occurs when an element is invalid
onreset	The event occurs when a form is reset
onsearch	The event occurs when the user writes something in a search field (for <input="search">)
onselect	The event occurs after the user selects some text (for <input> and <textarea>)
onsubmit	The event occurs when a form is submitted